Observational Program Calculi and the Correctness of Translations

Manfred Schmidt-Schauss, David Sabel, Joachim Niehren, Jan Schwinghammer

Observational Program Calculi and the Correctness of Translations

Manfred Schmidt-Schauß[a], David Sabel[a], Joachim Niehren[b], Jan Schwinghammer

[a] *Goethe-University Frankfurt am Main, Germany*
[b] *INRIA Lille, France, Links Project*

Abstract

For the issue of translations between programming languages with observational semantics, this paper clarifies the notions, the relevant questions, and the methods; it constructs a general framework, and provides several tools for proving various correctness properties of translations like adequacy and full abstractness, with a special emphasis on observational correctness. We will demonstrate that a wide range of programming languages and programming calculi and their translations can make advantageous use of our framework for focusing the analysis of their correctness.

Keywords: Contextual equivalence, correctness, semantics, translations

1. Introduction

Motivated by our experience in analyzing and proving properties of translations between programming languages with observational semantics, this paper clarifies the notions, the relevant questions, and the methods, and also constructs a general framework, and provides several tools for proving various correctness properties of translations like adequacy and full abstractness. The presented framework can directly be applied to the observational equivalences derived from the operational semantics of programming calculi (also with concurrency), to the relationship between specification and implementation, to the issue of correctness of embedding a language into another, and also to further issues in translations, and thus has a wide range of applications.

In order to be as general as possible, we use the term *observation* in this paper for *any* predicate on programs, *without* any further restriction like being a constructive one or an effectively computable property.

Motivation. Translating programs is an important operation in several fields of computer science. There are four main tasks where translations play an important role:

(1) Translation is the standard task of a compiler, where this is usually a conversion from a high-level language into an intermediate or low-level one, like an abstract machine language or an assembly-like language. Correctness of such a translation ensures correctness of the compiler.

Email address: sabel@ki.informatik.uni-frankfurt.de (David Sabel)

(2) Translations are required in programming languages for explaining the meaning of surface language constructs by decomposing them into a number of more primitive operations in the core part of the programming language. The translation paradigm is also useful for reasoning about the implementations of language extensions in terms of a core language (which are often packaged into the language's library). Typical examples are implementations of channels, buffers, or semaphores using a synchronizing primitive in the core part of the language, for example using mutable reference cells and futures in Alice ML [7, 40, 66], or using MVars in Concurrent Haskell [42]. Correctness of these implementations can be proved by interpreting the implementation as a translation from the extended language into the original language and then showing correctness of the translation.

(3) Optimization of programs by transforming them into programs of the same language, like inlining, partial evaluation and dead code removal are a special case of translations, where source and target language are the same. Again correctness of the program transformations can be seen correctness of translations.

(4) Translations are used to compare the expressiveness (and obtain corresponding expressiveness results) between different languages or programming models. Examples are showing adequacy or full abstractness of denotational models (e.g. [46, 31, 9, 2]), where the translation computes the denotation of the program, proving a language extension being conservative, or even showing non-expressiveness by proving the non-existence of "correct" translations (one such example is the non-encodability of the synchronous π-calculus in its asynchronous variant under mild restrictions [41]). A further example is the question for the expressive power of a sublanguage, viewed as embedded, and whether the language can be seen as a conservative extension of the sublanguage. An example scenario is removing syntactic sugar.

Correctness of these translations is an indispensable prerequisite for their safe use. However, there are various different views of the strength of correctness of a translation. This pluralism appears to be necessary and driven by practical needs, since the desired strength of correctness of a translation may depend on the specific setting.

From a bird's eye view a programming language is a set of programs $\mathcal{P}$ equipped with a notion of equivalence $\sim$ of programs (the semantics). A translation T maps programs from a source language $\mathcal{K}$ into a target language $\mathcal{K}'$. This suffices to define the commonly used notions of *adequacy* and *full abstractness*: The translation T is adequate iff for all (closed) programs $p_1, p_2 \in \mathcal{K}$ the implication $T(p_1) \sim_{\mathcal{K}'} T(p_2) \implies p_1 \sim_{\mathcal{K}} p_2$ holds; this can also be seen as the "no confusion" property, since it is equivalent to $p_1 \not\sim_{\mathcal{K}} p_2 \implies T(p_1) \not\sim_{\mathcal{K}'} T(p_2)$. If additionally $p_1 \sim_{\mathcal{K}} p_2 \implies T(p_1) \sim_{\mathcal{K}'} T(p_2)$ holds then T is fully abstract. Requiring fully abstract translations is often a too hard condition, while adequacy is a necessity. Without adequacy, in the target language equivalent programs may be interchanged correctly (since they are equivalent), but from the source level view, the semantics is changed. From a compilation point of view, full abstractness ensures that optimizations of programs can be

performed only in $\mathcal{K}$ or only in $\mathcal{K}'$ without missing opportunities. If a compilation is adequate, but not fully abstract (which is unavoidable in a majority of cases), then optimizing in $\mathcal{K}'$ and $\mathcal{K}$ is permitted, but optimizing only in $\mathcal{K}'$ may miss certain optimizations, since after compilation, some intentional knowledge may be lost.

Depending on the definitions of $\sim_{\mathcal{K}}$ and $\sim_{\mathcal{K}'}$, adequacy and even full abstractness may be too weak as a correctness notion. E.g., none of the properties ensures that programs before and after the translation have the same termination behavior, which is an inevitable requirement to conclude that T together with $\mathcal{K}'$ is a correct evaluator for $\mathcal{K}$. Thus we will be more concrete, and provide a general approach for program calculi with an operational semantics, which beside others covers the termination behavior of programs.

Results and Applications. We give a short and informal overview of the results and applications. Within the scope are (at least) programming languages where programs can be written down (using a syntax) and evaluated (using an operational semantics) and of several observations like successful evaluation in a program context (may-convergence, must-convergence). Syntactically, typing can be formulated and open and closed programs can be distinguished, and operationally, also concurrent systems, e.g. process calculi are in the scope.

A first useful result (where variants are already in the literature) is Theorem 3.16 which shows that there are easily checkable criteria for adequacy, namely compositionality and convergence equivalence. In this paper there are further extensions and variations of this method.

A general result concerning translations is Theorem 3.21 which exhibits conditions, when the translated programs are more or less equivalent to the original programs (in the image calculus); which is a theorem similar to an algebraic homomorphism theorem. It also provides a first result on conditions when a translation is an embedding.

Section 4 exhibits several conditions and scenarios for extending (resp. embedding) languages. Theorem 4.3 puts a common technique in semantics (approximation by $\bot$) for showing full abstraction into the context of our framework, where the approximations are represented as a family of translations.

Theorem 4.6 is a criterion for full abstractness under a minimal set of conditions. It complements the full abstractness criteria of Theorem 3.21. Corollaries 4.7 and 4.8 show that there are easy-to-check preconditions for applying Theorem 4.6.

Applications of the framework and the methods are in Section 5.2 and in Section 5.3. In particular Corollary 4.8 was applied in [38]. Another published result is the proof of conservativity of embedding the deterministic part of the language in the full language [55].

This paper presents a unifying approach for different variations of observational semantics and translations of programming languages. The usage may be as turnkey solution for proving properties of translations, and if this is not applicable, the usage will also be as a paradigm and guidance to clearly understand the real issues when analyzing the semantics and the correctness of translations for a programming language together with its evaluation.

Observational Program Calculi. For the formal semantics of a programming language several different approaches exist, e.g. there are equivalence notions based on the denota-

tional semantics, on logical relations, on bisimulations, and observational equivalence.

We will consider observational semantics for several reasons:

- Observational equivalence is based on the operational semantics and thus is naturally available for almost all programming languages. In case other semantics are available, the observational semantics is often the reference for various correctness criteria, like adequacy or full abstractness in denotational semantics. In addition, observational equivalences tend to be the maximal useful equivalences.

- A simple approach for defining a program equivalence is the extensional approach: compare the output of the programs (perhaps on all inputs) and request that the programs are equivalent iff the output is identical (the same value). However, this leads to lots of variations of definitions, and also might require extra testing abilities which might not be included in the expressive power of the languages. In contrast, for observational semantics the main criterion is convergence (or successful termination), which allows for a common notion of program equivalence. Moreover, usually observational semantics includes the above program equivalence defined by comparing outputs.

The notion of observational semantics is a syntactic approach and thus we will speak of programs, contexts, open and closed programs and convergence (i.e. successful termination), however, with an abstract meaning.

Observational semantics identifies programs if, and only if their successful termination (convergence) behavior is indistinguishable if one program is interchanged by the other one in any larger surrounding program. The most important instance of observational semantics is the well-known notion of *contextual equivalence* ([35, 46, 31, 10]): two programs p_1, p_2 are considered equivalent if they exhibit the same convergence behavior in all contexts C, denoted as $p_1 \sim p_2$. Note that it is usually not necessary to observe convergence to the same value, since the contexts of the underlying language have enough discrimination power to distinguish different values by convergence. However, for non-deterministic and concurrent programming languages a single (may-) convergence predicate $p{\downarrow_{\forall}}$ is insufficient, in the sense that the induced program equivalence does not distinguish programs that exhibit intuitively distinct behaviors. Instead, for non-deterministic and concurrent languages, a suitable equivalence arises from a combination of may- with must- or should-convergence (see e.g. [14, 15, 12, 53, 39, 61]). Accordingly, we will also consider an observational semantics which may be based on *multiple* convergence predicates. Observational semantics that consider convergence with some specific output, (for instance a name of a channel in barbed congruence in the π-calculus), could be modeled by (an infinite) set of convergence predicates (e.g. one for every channel name). If the languages are sufficiently expressive, these convergence predicates are in general undecidable and the derived relations are not recursively enumerable. Only may-convergence is recursively enumerable in several languages. Due to this undecidability and also due to the quantification over all contexts, establishing concrete contextual equivalences is usually not easy. However, various (in general incomplete) proof methods have been developed, general ones and calculus-specific ones. These include context lemmas (e.g., [31, 28, 25, 17, 62]), bisimulation methods (for instance,

[21, 20, 68, 45]), diagram-based methods (e.g., [26, 70, 39, 48, 47]), and characterizations of contextual equivalence in terms of logical relations (e.g., [44, 8, 3, 6]).

To express observational semantics for a broad class of programming languages, we define so-called observational program calculus. It abstracts from the details of a concrete programming language and – as we illustrate by examples – captures a lot of programming calculi, which are instances of an observational program calculus.

The main ingredients of an observational program calculus are programs, contexts, types, and convergence predicates (called *observation predicates* to stress the abstract level). However, convergence of programs is only tested for *closed* programs: This is more common in the literature, in particular for call-by-value languages, and also leads to more equations and thus a better language model. Contextual equivalence then only tests (open) programs in those contexts that close the programs. So we add further components to the observational program calculus: an abstract notion of *closedness*, which is represented as a subset of all programs, and an abstract notion of so-called *generalized closing substitutions*, which allow to close any program and are a subset of the contexts with specific properties. The latter component helps, among others, to ensure that contextual equivalence is a congruence.

Correctness of Translations. The goal of this paper is to present our approach on proving correctness of translations between two observational program calculi, which we applied for concrete and complex and quite different instances several times in the past (see Section 5.3), and will also lead to a better focus in future work. Thus we consider translations $T : \mathcal{K} \to \mathcal{K}'$ between source and target languages $\mathcal{K}$ and $\mathcal{K}'$ where both $\mathcal{K}$ and $\mathcal{K}'$ are observational program calculi. More specifically, in our setting T has to translate types and programs, but also the contexts and it maps observation predicates to observation predicates. One may argue that translating the contexts is a too strict requirement. However, for deeply comparing the observational semantics of $\mathcal{K}$ and $\mathcal{K}'$ this is inevitable. Even for non-contextual translations our framework may be helpful, since translations often can be split into a composition of several sub-translations such that several of the subtranslations are contextual where our framework is applicable.

Besides the notions of adequacy and full abstractness (where $\sim$ is contextual equivalence) we consider and examine further properties of the translation T which are important for its correctness. As explained before, it is a basic requirement that $\mathcal{K}'$ evaluates translated programs as they are evaluated in $\mathcal{K}$. This property is called *convergence equivalence* and means for all closed $\mathcal{K}$-programs p and all observation predicates $\downarrow_\forall$ of $\mathcal{K}$ that the equivalence $p{\downarrow_\forall} \iff T(p){\downarrow_{\forall T}}$ holds (where $\downarrow_{\forall T} = T(\downarrow_\forall)$).

If we add a weak form of compositionality of T as a requirement, then convergence equivalence results in our most important correctness notion: translation T is *observationally correct* if, and only if $C(p){\downarrow_\forall} \iff T(C)T(p){\downarrow_{\forall T}}$ for all $\mathcal{K}$-programs p, $\mathcal{K}$-contexts C, and $\mathcal{K}$-observation predicates $\downarrow_\forall$ (whenever $C(p)$ is closed). Observational correctness expresses that every testing of a source program (by a context together with an observation predicate) can compositionally be performed in $\mathcal{K}'$ without any difference. From a verification perspective we may view a context together with an observation predicate as a *specification*. In logical terms we may write $p \models (C, \downarrow_\forall)$ instead of $C[p]{\downarrow_\forall}$ to express that program p fulfills the

specification $(C, \downarrow_{\Diamond})$. Observational correctness could then be written as $p \models (C, \downarrow_{\Diamond})$ if and only if $T(p) \models (T(C), T(\downarrow_{\Diamond}))$ which emphasizes that the translation preserves and reflects the specification and that the specification itself must be translatable.

Observational correctness can also be expressed as convergence equivalence plus *compositionality up to observation*, where the latter means that $T(C(p))\downarrow_{\Diamond_T} \iff T(C)(T(p))\downarrow_{\Diamond_T}$. Compositionality up to observation is weaker than compositionality (i.e. $T(C(p)) = T(C)(T(p))$), and thus any compositional and convergence equivalent translation is observationally correct. As a consequence for compositional translations reasoning about contexts is not necessary to prove observational correctness.

As we show (see Proposition 3.16), a consequence of observational correctness is adequacy, i.e. once observational correctness of T is proved, we get for free that T is adequate. Note that the reverse implication does not hold (see Proposition 3.18).

Full abstractness of T often does not hold, since the target calculus $\mathcal{K}'$ has more contexts than $\mathcal{K}$ and thus translated programs can be distinguished by these contexts. However, in Section 4 we will also examine when and how full abstractness can be obtained for observationally correct translations provided $\mathcal{K}'$ can be embedded in $\mathcal{K}$, which is typically the case if $\mathcal{K}$ extends $\mathcal{K}'$ by new language constructs.

Applications. To illustrate our framework and demonstrate its applicability, we will point to rather diverse examples. During the definition of the framework and the properties of the translations, we use as running examples mostly variations of PCF as a well-known example of a typed lambda calculus both in its call-by-value and call-by-name variants.

As a fully worked out example we consider the Church encoding of pairs in a call-by-value lambda calculus; this example shows that our basic requirement for correctness, convergence equivalence, *fails* without an appropriate notion of typing. This example is important, since the typing issues raised by the encoding of pairs is an instance of the common situation where an abstract data type is implemented in terms of some operations on a given type.

These worked-out examples are rather small, but in a lot of works we applied the techniques represented in this paper to larger – real world – examples. In Section 5.3 we will give an overview of the quite different applications of the framework. We will also discuss other related work and how it is related to our abstract notions and which questions are addressed in the terms of our framework.

Overview. The paper is structured as follows: In Section 2 we present our notion of observational program calculi and show how to define the observational semantics for them. In Section 3 we define the notion of a translation between observational program calculi, introduce and discuss the fundamental properties of translations, and finally show some relations between these properties. In Section 4 we consider the specific case of language extensions and analyze the conditions under which full abstractness of embeddings and encodings can be deduced. In Section 5 we first consider some specializations of our framework, provide the observational correctness proof of Church's encoding of pairs, and we give an overview of larger examples where we applied the techniques of our framework. The discussion of related work is deferred to Section 6. We conclude in Section 7.

2. Observational Program Calculi

Our objective is to introduce a general notion of an observational program calculus that provides all ingredients for defining the observational semantics of a programming language in a systematic manner.

2.1. Starting Example

As a prototypical example for a sequential functional language, we present variants of PCF [46, 43]. We will also extend it with a nondeterministic choice operator, in order to illustrate the methods for treating concurrent programming languages.

The simply typed lambda calculus PCF_{cbv}. This is a call-by-value lambda calculus with fixed point operators, Booleans, and natural numbers. It can also be obtained by making Plotkin's language PCF [46, 43] call-by-value. The *types* τ are either the base types o for Booleans and ι for natural numbers, or function types recursively constructed from the base types, that is $\tau ::= o \mid \iota \mid \tau_1 \to \tau_2$. We assume an infinite set of variables ranged over by x, y, z and assume that every variable x has a predefined type $\Gamma(x)$. The *programs* are well-typed expressions p built from variables, constants for the Boolean values and all non-negative numbers $i \in \mathbb{N}$, abstraction and application, conditionals, function constants for arithmetic operations, a family of fixed point operators $\mathbf{fix}_\tau$ for all types τ of the form $((\tau_1 \to \tau_2) \to \tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)$. The *values* v of PCF_{cbv} are variables, constants, and lambda-abstractions.

$$\begin{array}{lcl} v & ::= & x \mid b \mid i \mid \mathbf{succ} \mid \mathbf{pred} \mid \mathbf{zero?} \mid \mathbf{fix}_\tau \mid \lambda x.p \\ p & ::= & v \mid (p_1\ p_2) \mid \mathbf{if}\ p\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2 \end{array}$$

A program is *closed* if it does not contain free variables, i.e., if every occurrence of a variable is in the scope of some lambda-binder. We only consider well-typed programs. Every well-typed program has a unique type, since we assume that all variables have a unique type (given by Γ). The typing rules are omitted, since these are standard. A *context* C is like a program p except that it contains a single occurrence of a "hole". We assume that all holes are annotated by a type and write $[.]_\tau$ for the hole of type τ. The program obtained by replacing the hole of context C by a program p of the same type is denoted by $C(p)$. A context C of type τ' with a hole of type τ can thus be identified with a function that maps programs of type τ to programs of type τ'. Note that the free variables in p may be bound in $C(p)$, so that $C(p)$ may become closed even though p was not.

We next define the evaluation in PCF_{cbv} by a small-step operational semantics $p \to p'$. The possible reductions are defined below:

$$\begin{array}{ll} (\lambda x.p)\, v \to p[v/x] & (\mathbf{succ}\ i) \to i+1 \\ \mathbf{fix}_\tau\ \lambda x.p \to p[\lambda y.(\mathbf{fix}_\tau\ \lambda x.p)\, y/x] & (\mathbf{pred}\ i) \to i-1 \text{ if } i > 0 \\ (\mathbf{if\ true\ then}\ p_1\ \mathbf{else}\ p_2) \to p_1 & (\mathbf{zero?}\ 0) \to \mathbf{true} \\ (\mathbf{if\ false\ then}\ p_1\ \mathbf{else}\ p_2) \to p_2 & (\mathbf{zero?}\ i) \to \mathbf{false} \text{ if } i > 0 \end{array}$$

These reduction rules can be performed in all reduction contexts R defined as follows:

$$R ::= [.]_\tau \mid (R\ p) \mid (v\ R) \mid \mathbf{if}\ R\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2$$

The standard reduction is $R(p) \to R(p')$, if $p \to p'$ by one of the reductions above. We say that a program p converges and write $p{\downarrow_{\text{PCF}cbv}}$ if there exists a value v, such that $p \to^* v$. Otherwise we say that p diverges and write $p{\uparrow_{\text{PCF}cbv}}$. Note that programs of the form $R(\mathbf{pred}\ 0)$ are irreducible even though they are not a value, and thus such (erroneous) programs diverge, thus they represent diverging programs, sometimes written $\bot$. Other divergent programs are non-terminating programs. There are some nonterminating programs in PCF_{cbv} even though we impose simple typing, since PCF_{cbv} admits fixed point operators. Moreover, we can define a non-terminating closed program for every type τ as $\Omega_\tau := (\mathbf{fix}_{\tau'}\ (\lambda x.x))\ 0$, where $\tau' = ((\iota \to \tau) \to \iota \to \tau) \to (\iota \to \tau)$ and x is a variable with $\Gamma(x) = (\iota \to \tau)$.

Contextual equivalence for programs p and p' of type τ is defined by observing termination in all *closing* contexts, i.e. $p \sim_\tau p'$ iff $C(p){\downarrow_{\text{PCF}cbv}} \Leftrightarrow C(p'){\downarrow_{\text{PCF}cbv}}$ for all appropriately typed contexts C closing p and p'. This captures the intuition that p and p' may be freely exchanged in any larger closed program without affecting its observable behavior. For instance, $\mathbf{if}\ x\ \mathbf{then\ true\ else\ true} \sim_o \mathbf{true}$ holds in PCF_{cbv}. The intuition is that free variables in a call-by-value language can only be substituted by correctly typed values, so that x must be instantiated by **true** or **false**. The instantiation can be done by reduction in contexts such as $(\lambda x.[.]_o)\ p$ where p converges to some Boolean. Since the argument p must be evaluated before call-by-value beta reduction can be applied, x can only be substituted by a Boolean this way. Note, however, that if also non-closing contexts were permitted in the definition of contextual equivalence, then the identity context $[.]_o$ could be used to distinguish both expressions.

Extension to $\text{PCF}_{cbv,\oplus}$ *with nondeterministic choice.* More evolved concurrent programming languages can be obtained by extending sequential programming languages by communicating threads. In order to illustrate the main consequences for observational semantics, we consider a more modest extension $\text{PCF}_{cbv,\oplus}$ which adds nondeterministic choice to PCF_{cbv}.

$\text{PCF}_{cbv,\oplus}$ extends PCF_{cbv} by a family of constants $\mathbf{choice}_{\tau\to\tau\to\tau}$ for any type τ. The additional reduction axioms are $\mathbf{choice}_\tau\ v_1\ v_2 \to v_1$ and $\mathbf{choice}_\tau\ v_1\ v_2 \to v_2$. For such nondeterministic languages it is not sufficient to observe only whether a program *may* terminate, but also termination properties in all computation paths. *May-convergence* can be defined in the same way as the observation predicate ${\downarrow_{\text{PCF}cbv}}$ before, but cannot be understood as *convergence* any more. Its negation ${\uparrow_{\text{PCF}cbv}}$ has now to be read as *must-divergence*. In addition, we observe so-called *should-convergence*[1] where a program p should-converges ($p{\Downarrow_{\text{PCF}cbv}}$) if it is impossible to reduce p to a must-divergent expression, i.e. $p{\Downarrow_{\text{PCF}cbv}}$ iff for every p' the implication $p \to^* p' \implies p'{\downarrow_{\text{PCF}cbv}}$ holds.

The contextual equivalence is now defined with respect to both may- and should-convergence, i.e., $p \sim_\tau p'$ holds for two expressions p and p' of type τ if for all appropriately typed contexts C closing p and p': $C(p){\downarrow_{\text{PCF}cbv}} \Leftrightarrow C(p'){\downarrow_{\text{PCF}cbv}}$ and $C(p){\Downarrow_{\text{PCF}cbv}} \Leftrightarrow C(p'){\Downarrow_{\text{PCF}cbv}}$. For instance, if $\Gamma(x) = \tau$, then we obtain $\mathbf{choice}_{\tau\to\tau}\ (\lambda x.x)\ (\lambda x.\Omega_\tau) \not\sim_{\tau\to\tau} \lambda x.x$,

[1]Note that should-convergence does not exclude weak-divergences. Sometimes it is also called must-convergence. A different notion of must-convergence in the literature is obtained by defining $p{\Downarrow\!\!\Downarrow_{\text{PCF}cbv}}$ such that there is no infinite reduction sequence starting from p and every reduction sequence ends in a value.

since should-convergence in the context $([.]_{\tau\to\tau}\ v)$ for a value v of type τ distinguishes these two expressions (in contrast to may-convergence). As in deterministic PCF_{cbv}, the contextual equivalence in $\text{PCF}_{cbv,\oplus}$ is a typed congruence relation, as we will show in Proposition 2.6.

2.2. Observational Semantics

In order to develop observational semantics for various programming languages in a uniform framework, we need an abstract notion of a program calculus that abstracts from various kinds of *typed programs*, *typed contexts*, *observation predicates* and *closed programs*.

Definition 2.1. An *observational program pre-calculus* is a tuple $(\mathcal{P}, \mathsf{Clos}, \mathcal{C}, \mathcal{O}, \mathcal{T}, \mathsf{type})$ where:

- $\mathcal{P}$ is a set of *programs* ranged over by p.
- Clos a subset of programs that are called *closed*.
- $\mathcal{C}$ is a set of *contexts* ranged over by C.
- $\mathcal{O}$ is a set of predicates $\downarrow_{\forall} : \mathcal{P} \to \mathbb{B}$ called *observation predicates*. For convenience, we write $p{\downarrow_{\forall}}$ for the application of $\downarrow_{\forall}$ to a program p. Furthermore, we write $\uparrow$ for its negation, i.e. $p{\uparrow}$ iff $p{\downarrow_{\forall}}$ does not hold.
- $\mathcal{T}$ is a set of *types* ranged over by τ, and
- type is a relation with $\mathsf{type} \subseteq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{C} \times (\mathcal{T} \times \mathcal{T}))$.

In the notation we will also use type as a set valued function, and write $\mathsf{type}(p) = \{\tau \mid (p,\tau) \in \mathsf{type}\} \subseteq 2^{\mathcal{T}}$ for the set of types of a program p and $\mathsf{type}(C) = \{(\tau_1,\tau_2) \mid (C,\tau_1,\tau_2) \in \mathsf{type}\} \subseteq 2^{\mathcal{T}\times\mathcal{T}}$ for the set of types of a context C. For all types τ, τ_1, τ_2 we denote by $\mathcal{P}_\tau$ the set of all programs of type τ and by $\mathcal{C}_{\tau_1,\tau_2}$ the set of all contexts of type (τ_1,τ_2).

We assume that every program and context has at least one type, i.e., $\mathcal{P} = \{p \mid p \in \mathcal{P}_\tau, \tau \in \mathcal{T}\}$, and $\mathcal{C} = \{C \mid C \in \mathcal{C}_{\tau_1,\tau_2}, \tau_1 \in \mathcal{T}, \tau_2 \in \mathcal{T}\}$. We also assume that every context $C \in \mathcal{C}$ is a partial function $C : \mathcal{P} \to \mathcal{P}$ that is type-correct and closed under composition, i.e.:

- for all types $\tau_1, \tau_2 \in \mathcal{T}$, all contexts $C \in \mathcal{C}_{\tau_1,\tau_2}$, and programs $p \in \mathcal{P}_{\tau_1}$: $C(p) \in \mathcal{P}_{\tau_2}$, and
- for all types $\tau_1, \tau_2, \tau_3 \in \mathcal{T}$, and contexts $C \in \mathcal{C}_{\tau_1,\tau_2}, C' \in \mathcal{C}_{\tau_2,\tau_3}$: $C' \circ C \in \mathcal{C}_{\tau_1,\tau_3}$. □

Even though observational program pre-calculi are typed, untyped calculi do also fit into our framework, since one can choose $\mathcal{T}$ to be a singleton containing the 'universal' type. In this case, the set of contexts is a semigroup. It is possible to describe calculi as an observational program pre-calculus, where programs may have several types in which case type is a proper relation. An example is a "polymorphic" PCF-variant where **fix** is a single constant with an infinite number of types, and where the identity $\lambda x.x$ has an infinite number of types of the form $\tau \to \tau$. In this calculus, $\lambda x.x \sim_{o\to o} \lambda y.y$ as well as $\lambda x.x \sim_{\iota\to\iota} \lambda y.y$

holds. Another example would be a calculus with an ordering on the types, i.e. there may be supertypes and subtypes, which can be modelled by assigning several types to expressions.

Including the component Clos is necessary to define an observational equivalence on open programs, while only taking the convergence behavior in closing contexts into account. As already argued before, this is a necessity for call-by-value calculi.

Example 2.2. *The call-by-value lambda calculus* PCF_{cbv} *from Section 2.1 matches the definition of an observational program pre-calculus as follows. The set* $\mathcal{T}$ *contains all simple types* τ*, the set of* $\mathcal{P}$ *all well-typed lambda expressions* p*, the set* Clos *all closed well-typed lambda expressions, and the set* $\mathcal{C}$ *all well-typed contexts* C*. The function* type *maps a program* p *to its unique type* τ*, and a context* C *of type* τ *and a hole of type* τ' *to the pair* (τ', τ)*. The set of observation predicates is* $\mathcal{O} = \{\downarrow_{\mathrm{PCF}cbv}\}$*.*

The extended call-by-value lambda calculus $\mathrm{PCF}_{cbv,\oplus}$ *also matches the definition of an observational program pre-calculus. Programs and contexts may now contain the choice operator in addition, so that the function* type *needs to be extended too, as well as the set of closed programs. The most important difference is the set of observation predicates* $\mathcal{O} = \{\downarrow_{\mathrm{PCF}cbv}, \Downarrow_{\mathrm{PCF}cbv}\}$*, which are needed to account for nondeterminism properly.*

We next define an observational equivalence that can be defined for any observational program pre-calculus. We will derive it from an observational preorder that allows more flexibility, and is an analogue of the domain-theoretic information preorder:

Definition 2.3. Let $\mathcal{K}$ be an observational program pre-calculus. For any type τ of $\mathcal{K}$, we define the following binary relations for all programs p_1, p_2 of type τ and $\downarrow_{\diamond} \in \mathcal{O}$:

Observational preorders

- $p_1 \leq_{\downarrow_{\diamond},\tau} p_2$ iff for all types τ' and contexts $C \in \mathcal{C}_{\tau,\tau'}$ closing p_1 and p_2: $C(p_1)\downarrow_{\diamond} \Rightarrow C(p_2)\downarrow_{\diamond}$,
- $p_1 \leq_{\tau} p_2$ iff for all observation predicates $\downarrow_{\diamond} \in \mathcal{O}$: $p_1 \leq_{\downarrow_{\diamond},\tau} p_2$.

Observational equivalences

- $p_1 \sim_{\downarrow_{\diamond},\tau} p_2$ iff $p_1 \leq_{\downarrow_{\diamond},\tau} p_2$ and $p_2 \leq_{\downarrow_{\diamond},\tau} p_1$.
- $p_1 \sim_{\tau} p_2$ iff $p_1 \leq_{\tau} p_2$ and $p_2 \leq_{\tau} p_1$.

Note that it does not make a difference if the observational equivalence would be defined as: $p_1 \sim_{\downarrow_{\diamond},\tau} p_2$ iff for all types τ' and contexts $C \in \mathcal{C}_{\tau,\tau'}$ closing p_1 and p_2: $C(p_1)\downarrow_{\diamond} \Leftrightarrow C(p_2)\downarrow_{\diamond}$.

2.3. Congruences

We present a general restriction on observational program pre-calculi to ensure that its observational equivalence becomes a congruence. The idea is to impose the existence of a set of generalized closing substitutions that can be applied to non-closed programs so that they become comparable to other closed programs. These substitutions should be chosen similarly to substitutions for the lambda calculus replacing free variables by closed expressions.

Definition 2.4. An *observational program calculus* is a tuple $(\mathcal{P}, \mathsf{Clos}, \mathcal{C}, \mathcal{O}, \mathcal{T}, \mathsf{type}, \mathcal{S})$ such that $(\mathcal{P}, \mathsf{Clos}, \mathcal{C}, \mathcal{O}, \mathcal{T}, \mathsf{type})$ is an observational program pre-calculus and $\mathcal{S} \subseteq \mathcal{C}$ a subset of contexts, called *generalized closing substitutions*. We will write $\mathcal{S}_{\tau,\tau'}$ for the set of generalized closing substitutions of type (τ, τ'), and require that any observational program calculus satisfies the following properties for all types $\tau, \tau', \tau'' \in \mathcal{T}$:

1. $\mathcal{S}$ is closed under typed function composition, i.e. for all $C \in \mathcal{S}_{\tau,\tau'}$, $C' \in \mathcal{S}_{\tau',\tau''}$: $C' \circ C \in \mathcal{S}_{\tau,\tau''}$, and

2. for all $p \in \mathcal{P}_\tau$ there is some type τ' and some generalized closing substitution $C \in \mathcal{S}_{\tau,\tau'}$, such that $C(p)$ is closed, and

3. for all $p \in \mathcal{P}_\tau$ and generalized closing substitutions $C \in \mathcal{S}_{\tau,\tau'}$: if p is closed, then also $C(p)$ is closed, and

4. for all closed programs $p \in \mathcal{P}_\tau$, generalized closing substitutions $C \in \mathcal{S}_{\tau,\tau'}$, and observation predicates $\downarrow_\diamond$ in $\mathcal{O}$: $C(p)\downarrow_\diamond \iff p\downarrow_\diamond$. □

For call-by-value lambda calculi such as PCF_{cbv}, an appropriate choice of the set of generalized closing substitutions $\mathcal{S}$ would be compositions of all appropriately typed contexts $(\lambda x.[.]_\tau)\ v$ where v is a closed value. On the other hand, if $\mathcal{S}$ would be chosen as all closed contexts, then condition (4) of Definition 2.4 is in general false, since for example in PCF_{cbv}: $\mathbf{true}\downarrow_{\text{PCF}cbv}$, and for $D := (\Omega_{o\to o}\ [.]_o)$, we have $D(\mathbf{true})\uparrow_{\text{PCF}cbv}$. Note also that condition (2) implies that $\forall\tau.\exists\tau' : \mathcal{S}_{\tau,\tau'} \neq \emptyset$.

Lemma 2.5. *For an observational program calculus $\mathcal{K}$ the following holds: For any type τ and finite subset $Q \subseteq \mathcal{P}_\tau$, there is some type τ' and some generalized closing substitution $C \in \mathcal{S}_{\tau,\tau'}$, such that $C(p)$ is closed for all $p \in Q$.*

Proof. The proof is by induction on $|Q|$. The base case where $|Q| = 1$ is condition 2 of Definition 2.4. Let $Q = \{p_1, \ldots, p_n\} \subseteq \mathcal{P}_\tau$ where $n \geq 2$. Then by condition 2.4.(2) there is $C_1 \in \mathcal{S}_{\tau,\tau'}$, such that $C_1(p_1)$ is closed. The induction hypothesis applied to $\{C_1(p_2), \ldots, C_1(p_n)\}$ shows that there is a type τ'' and $C_2 \in \mathcal{S}_{\tau,\tau''}$, such that $C_2(C_1(p_i))$ for $i = 2, \ldots, n$ is closed. By condition 2.4.(3), also $C_2(C_1(p_1))$ is closed. Now we apply condition 2.4.(1), which shows that $C_2 \circ C_1 \in \mathcal{S}_{\tau,\tau''}$. □

Proposition 2.6. *For any observational program calculus $(\mathcal{P}, \mathsf{Clos}, \mathcal{C}, \mathcal{O}, \mathcal{T}, \mathsf{type}, \mathcal{S})$, type τ in $\mathcal{T}$ and observation predicate $\downarrow_\diamond$ in $\mathcal{O}$, the relations $\leq_{\downarrow,\tau}$ and $\leq_\tau$ are* typed precongruences *and the relations $\sim_{\downarrow,\tau}$ and $\sim_\tau$ are* typed congruences. *That is:*

- *The relations $\leq_{\downarrow,\tau}$ and $\leq_\tau$ are* typed precongruences*, i.e. they are preorders and for all programs $p_1, p_2 \in \mathcal{P}_\tau$, all types $\tau' \in \mathcal{T}$, all contexts $C \in \mathcal{C}_{\tau,\tau'}$: $p_1 \leq_{\downarrow,\tau} p_2 \Rightarrow C(p_1) \leq_{\downarrow,\tau'} C(p_2)$ and in analogy: $p_1 \leq_\tau p_2 \Rightarrow C(p_1) \leq_{\tau'} C(p_2)$.*

- *The relations $\sim_{\downarrow,\tau}$ and $\sim_\tau$ are* typed congruences*, i.e. they are typed precongruences and equivalence relations.*

Proof. We only consider the observational preorder, which implies the part for observational equivalence. It is easy to see that $\leq_{\downarrow,\tau}$ is reflexive. In order to check that each $\leq_{\downarrow,\tau}$ is transitive, let $p_1 \leq_{\downarrow,\tau} p_2 \leq_{\downarrow,\tau} p_3$ and C be a context in $\mathcal{C}_{\tau,\tau'}$, such that $C(p_1)$ and $C(p_3)$ are closed, and such that $C(p_1)\downarrow_{\downarrow}$. We have to show that $C(p_3)\downarrow_{\downarrow}$. If $C(p_2)$ is closed, then $C(p_1)\downarrow_{\downarrow}$ implies $C(p_2)\downarrow_{\downarrow}$, which in turn implies $C(p_3)\downarrow_{\downarrow}$. In the other case, where $C(p_2)$ is not closed, Lemma 2.5 implies that there is a type τ'' and a generalized closing substitution $D \in \mathcal{S}_{\tau',\tau''}$ such that $D(C(p_i))$ for $i = 1, 2, 3$ are closed. Condition (4) of Definition 2.4 shows that $D(C(p_1))\downarrow_{\downarrow} \iff C(p_1)\downarrow_{\downarrow}$ and that $D(C(p_3))\downarrow_{\downarrow} \iff C(p_3)\downarrow_{\downarrow}$, hence $D(C(p_1))\downarrow_{\downarrow}$. Since $D \circ C$ is also a context by Definition 2.1, we obtain $D(C(p_2))\downarrow_{\downarrow}$ (from $p_1 \leq_{\downarrow,\tau} p_2$), which in turn implies also $D(C(p_3))\downarrow_{\downarrow}$ (since $p_2 \leq_{\downarrow,\tau} p_3$ and since $D(C(p_3))$ is closed), and thus $C(p_3)\downarrow_{\downarrow}$.

It remains to show that $\leq_{\downarrow,\tau}$ is compatible with contexts: Let $p_1 \leq_{\downarrow,\tau} p_2$ and $C \in \mathcal{C}_{\tau,\tau'}$. For any context $C' \in \mathcal{C}_{\tau',\tau''}$ where $C'(C(p_1))$ and $C'(C(p_2))$ are closed, the inequation $p_1 \leq_{\downarrow,\tau} p_2$ obviously implies that $C'(C(p_1))\downarrow_{\downarrow} \implies C'(C(p_2))\downarrow_{\downarrow}$, since $C' \circ C$ is also a context. □

A further consequence of the restriction on observational program pre-calculi is that observationally equivalent, closed programs have the same observations:

Lemma 2.7. *Let p_1, p_2 be two closed programs of an observational program calculus $\mathcal{K}$ of the same type τ and $\downarrow$ an observation predicate of $\mathcal{K}$. If $p_1\downarrow_{\downarrow}$ and one of the following holds: $p_1 \sim_\tau p_2$, $p_1 \sim_{\downarrow,\tau} p_2$, $p_1 \leq_\tau p_2$, or $p_1 \leq_{\downarrow,\tau} p_2$, then also $p_2\downarrow_{\downarrow}$.*

Proof. Let p_1, p_2 be closed and $p_1\downarrow_{\downarrow}$. Then Lemma 2.5 and condition (4) of Definition 2.4 show that there is some $C \in \mathcal{S}$ such that $C(p_1)$ and $C(p_2)$ are closed, and $C(p_1)\downarrow_{\downarrow}$. Then every relation above implies $C(p_2)\downarrow_{\downarrow}$, and again the condition (4) shows that $p_2\downarrow_{\downarrow}$. □

In the following, types are sometimes omitted in the notation, and we implicitly assume that type information follows from the context.

2.4. Further Examples

We sketch some further examples to illustrate the range of situations that fit the definition of an observational program calculus. Many other lambda calculi fit into our framework, like the lazy lambda calculus or call-by-need lambda calculi. Observational Program Calculi can also capture models of concurrent computation, such as CCS or other process calculi. Also abstract machines fit into this framework, where machine environments, stacks, heaps etc. may be modelled as contexts. Given that observational program calculi do not rely on small-step semantics, also calculi with big-step semantics fit into our framework.

Example 2.8 (Call-by-name PCF)**.** *We consider the call-by-name lambda calculus PCF_{cbn}, which is a variant of the call-by-value lambda calculus PCF_{cbv} with the same expressions. The call-by-name beta-reduction is now applicable for any argument p': $(\lambda x.p)\, p' \to p[p'/x]$ and the fixpoint reduction is modified similarly: $\mathbf{fix}_\tau\ p \to p\ (\mathbf{fix}_\tau\ p)$. The reduction contexts are different, since they do not force evaluation of arguments, so they are:*

$$R ::= [.]_\tau \mid (R\ p') \mid \mathbf{if}\ R\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2 \mid \mathbf{pred}\ R \mid \mathbf{succ}\ R \mid \mathbf{zero?}\ R.$$

The generalized closing substitutions in $\mathcal{S}$ are the compositional closure of all appropriately typed contexts $((\lambda x.[.]_\tau)\ p')$ where p' is any closed expression.

In contrast to PCF_{cbv}*, now it does no longer make a difference if convergence is defined for all expressions, and contextual equivalence could be defined w.r.t. all (perhaps open) contexts. E.g., the two expressions* **if** x **then true else true** *and* **true** *are not observationally equivalent in* PCF_{cbn}*, since they can be distinguished by the context $C := (\lambda x.[.]_o)\ \Omega_o$: the program C[**true**] reduces to* **true** *while C[***if** x **then true else true**] *diverges. Thus, a surrounding context together with reduction may instantiate the free variables of an open program by* any *program, since* PCF_{cbn} *has a call-by-name reduction.*
This is different under call-by-value strategies, for example PCF_{cbv}*, where the closing contexts are composed of contexts $((\lambda x.[.]_\tau)\ p')$ where p' must be closed and converging.*

In particular, Definition 2.1 captures not only lambda calculi, but also process calculi:

Example 2.9 (CCS)**.** *CCS [32] may be viewed as an (untyped) observational program calculus: for a fixed action set Σ, both programs and contexts are given by the set of CCS processes $P, Q, \ldots$, and $P \circ Q$ as well as $P(Q)$ are given by the parallel composition $P \mid Q$. More precisely, contexts are given by the functions f_P with $f_P(Q) = P \mid Q$. Since there are no variables, we define $\mathsf{Clos} = \mathcal{P}$, and generalized closing substitutions $\mathcal{S} = \{id\}$ where id is the identity. By considering observation predicates $\downarrow_{CCS,\sigma}$ for every $\sigma \in \Sigma^\omega$ such that $P{\downarrow_{CCS,\sigma}}$ holds if σ is a trace of P, we obtain a trace-based testing equivalence $\sim$ on processes. Variations are possible, e.g. by restricting the observations to finite traces $\sigma \in \Sigma^*$ (see [37]).*

The terms "calculus" and "observation" in Definition 2.1 are to be understood in a loose sense, especially "observable" does not mean that the predicates are effectively computable or even recursively enumerable. Hence our notion of an observational program calculus is very general, and for instance, semantic models also fit the definition of an admissible observational program calculus:

Example 2.10 (CPOs)**.** *A semantic counterpart to call-by-value PCF is given by ω-complete pointed partial orders (cppos) and continuous maps. More precisely, if $\mathcal{D}_B$ and $\mathcal{D}_N$ are the flat cppos with underlying sets $\{0, 1\}$ and $\mathbb{Z}$ respectively, we let $\mathcal{D}_{\tau_1 \to \tau_2} = (\mathcal{D}_{\tau_1} \to \mathcal{D}_{\tau_2})_\perp$ be the set of strict continuous functions from $\mathcal{D}_{\tau_1}$ to $\mathcal{D}_{\tau_2}$ extended with a new least element, and order $\mathcal{D}_{\tau_1 \to \tau_2}$ pointwise. We can then take $\mathcal{P}_\tau$ to be the underlying set of $\mathcal{D}_\tau$. Since there are no variables, we choose $\mathsf{Clos} = \mathcal{P}$. The contexts are continuous maps, i.e., $\mathcal{C}_{\tau_1,\tau_2} = \mathcal{D}_{\tau_1} \to \mathcal{D}_{\tau_2}$, and for $a \in \mathcal{P}_\tau$ the observation $a{\downarrow_{cpo}}$ holds if $a \neq \perp$. Since every program is closed, we set $\mathcal{S} := \{id_\tau \mid \text{for any type } \tau\}$ where id_τ is the identity on type τ. In this example, $a \sim_\tau a'$ if and only if $a = a'$.*

3. Translations

A translation is a mapping between observational program calculi that often arises very concretely when relating two programming languages. Examples are compilations of one programming language into another one, which may induce a mapping between possibly

rather different calculi, or the removal of syntactic sugar, which may be expressed as a mapping from an extended calculus into a core calculus, or the embedding of a calculus into its extended version. Furthermore, expressivity results between different programming languages are usually obtained by mapping one programming language into another one.

As a simple concrete example let us consider the removal of Booleans in PCF_{cbv}, i.e. $\text{PCF}_{cbv,-\mathbb{B}}$ is the calculus PCF_{cbv} where types do not contain o, there are no Boolean values, the constant **zero**? is not included and conditionals **if** p_1 **then** p_2 **else** p_3 require p_1 to be of type ι. The reduction axioms for conditionals are replaced by

$$(\textbf{if } i \textbf{ then } p_1 \textbf{ else } p_2) \to p_1 \text{ if } i > 0 \qquad\qquad (\textbf{if } 0 \textbf{ then } p_1 \textbf{ else } p_2) \to p_2$$

The questions that arise are: Is there an encoding from $\text{PCF}_{cbv} \to \text{PCF}_{cbv,-\mathbb{B}}$? In which sense are such translations (of course, also even more complex ones) between observational program calculi correct, i.e., how does the semantics in source and target calculus relate with respect to the translation, and what are the correctness requirements?

3.1. Pre-Translations

From a very general view, a translation is a mapping from types to types and a type correct mapping from programs to programs. Let $\mathcal{K}$ and $\mathcal{K}'$ be two observational program pre-calculi with types $\mathcal{T}$ and $\mathcal{T}'$, programs $\mathcal{P}$, $\mathcal{P}'$, and observational preorders $\leq_\tau$ and $\leq'_{\tau'}$.

Definition 3.1. A pre-translation T from an observational program pre-calculus $\mathcal{K}$ to an observational program pre-calculus $\mathcal{K}'$ is a function which maps types to types and programs to programs, i.e. $T : \mathcal{T} \to \mathcal{T}'$ and $T : \mathcal{P} \to \mathcal{P}'$ such that for all $p \in \mathcal{P}_\tau$: $T(p) \in \mathcal{P}_{T(\tau)}$.

Already for pre-translations we can define *adequacy*, *full abstractness* and the *isomorphism property* as correctness notions. Both properties speak about the relation between the observational preorders of the source and the target calculus of a pre-translation. These properties are quite standard and sometimes are also used for relating other formalisms, e.g. a contextual semantics and denotational semantics.

Definition 3.2. We call a pre-translation T from $\mathcal{K}$ to $\mathcal{K}'$

- **adequate** if T is $\leq$-reflecting, i.e., for all types $\tau \in \mathcal{T}$ and programs $p_1, p_2 \in \mathcal{P}_\tau$: $T(p_1) \leq'_{T(\tau)} T(p_2) \implies p_1 \leq_\tau p_2$,
- **fully abstract** if T is $\leq$-preserving and $\leq$-reflecting; i.e., for all types $\tau \in \mathcal{T}$ and programs $p_1, p_2 \in \mathcal{P}_\tau$: $p_1 \leq_\tau p_2 \iff T(p_1) \leq'_{T(\tau)} T(p_2)$,
- **an isomorphism** if T is a bijection on the types, T is a bijection between $\mathcal{P}/\sim$ and $\mathcal{P}'/\sim'$, and T is fully abstract.

Example 3.3. *Simple examples for pre-translations are embeddings like the identity translation T_{incl} from PCF_{cbv} into $\text{PCF}_{cbv,\oplus}$ defined by $T_{incl}(\tau) = \tau$ and $T_{incl}(p) = p$. One may*

expect that T_{incl} is fully abstract (but not an isomorphism). However, the (direct) proof is non-trivial, since it requires to reason about all contexts in $\mathrm{PCF}_{cbv,\oplus}$.

As another example we define a pre-translation $T_{-\mathbb{B}}$ from PCF_{cbv} *into* $\mathrm{PCF}_{cbv,-\mathbb{B}}$ *as follows: On the types, $T_{-\mathbb{B}}$ replaces all occurrences of o by ι, and on programs, $T_{-\mathbb{B}}$ is defined as $T_{-\mathbb{B}}(\mathbf{true}) = 1$, $T_{-\mathbb{B}}(\mathbf{false}) = 0$, $T_{-\mathbb{B}}(\mathbf{zero}?) = \lambda x.\mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ 1$; $T_{-\mathbb{B}}(\mathbf{fix}_\tau) = \mathbf{fix}_{T_{-\mathbb{B}}(\tau)}$ and applied homomorphically to all other language constructs. Clearly, $T_{-\mathbb{B}}$ is not an isomorphism (since $T_{-\mathbb{B}}$ is not bijective on types). $T_{-\mathbb{B}}$ is adequate (see Example 3.9 and Theorem 3.16). However, it is not fully abstract, since e.g. for $p_1 := \lambda x.x$ and $p_2 = \lambda x.\mathbf{if}\ x\ \mathbf{then}\ \mathbf{true}\ \mathbf{else}\ \mathbf{false}$ the equation $p_1 \sim_{o\to o} p_2$ holds in* PCF_{cbv}, *but $T_{-\mathbb{B}}(p_1) \not\sim_{\iota\to\iota} T_{-\mathbb{B}}(p_2)$ in* $\mathrm{PCF}_{cbv,-\mathbb{B}}$.

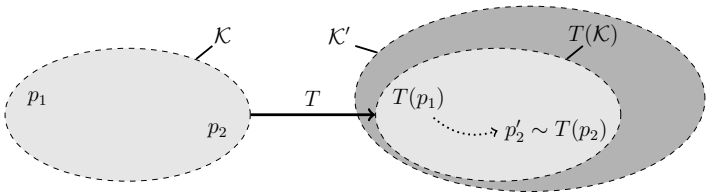
Note that the notion of translation and isomorphism is consistent with viewing observational program pre-calculi and pre-translations as a category. Every observational program pre-calculus has Id as an isomorphism, and an isomorphism between $\mathcal{K}$ and $\mathcal{K}'$ has an inverse translation that composes to the identity on $\mathcal{K}$ (or $\mathcal{K}'$, respectively).

One obvious consequence of the definition is that all three properties are closed w.r.t. composition of translations:

Proposition 3.4. *Let $\mathcal{K}, \mathcal{K}', \mathcal{K}''$ be program calculi and P be one of the properties of pre-translations, adequacy, full abstractness or being an isomorphism. If $T : \mathcal{K} \to \mathcal{K}'$ and $T' : \mathcal{K}' \to \mathcal{K}''$ are pre-translations with property P then functional composition $T' \circ T : \mathcal{K} \to \mathcal{K}''$ is also a pre-translation with property P.*

This proposition allows us to prove adequacy (or full abstractness) of a pre-translation by decomposing it and proving the adequacy (or full abstractness) of all of its parts.

Before introducing further methods and properties, let us discuss in which case a translation should be called correct. More generally, if we view a translation as a compilation of one calculus into another, where the intuition is to compile programs of a high level language $\mathcal{K}$ into a low level language $\mathcal{K}'$. When would we accept a compilation (translation) as correct?



One minimal sensible correctness requirement is that convergence of programs is unchanged; a second one is that testing a program and its result "through the compiler" must be the same as within the high-level language itself; a third requirement is that if $T(p_1)$ is the result of a compilation of p_1, and p_2' is the result of optimizing the program $T(p_1)$ in the low level language where $p_2' \sim T(p_2)$ for some $\mathcal{K}$-program p_2, then p_2 must be equivalent to p_1 in the high level language. The third requirement is the already introduced notion of adequacy. A compilation of a high-level program into a rather low level programming language is usually not fully abstract, since it is possible to test the implemented program or function for implementation details, which is impossible in the high-level language, and

which may lead to the counter-intuitive effect that programs (functions) p_1, p_2 that are indistinguishable in the high-level language are observably different after their compilation. This is often called a loss of knowledge or of intention of the programmer, but this is in general unavoidable, since the low level language can see and observe the implementation details. This can be partly overcome by optimizing the programs using the semantics of the high-level language.[2]

The first requirement is called *convergence equivalence* (see (ce) in Def. 3.8), and the first and second together are exactly the property of *observational correctness* (see (oc) in Def. 3.8 and its alternative characterization (acooc) in Lemma 3.12). Neither of these notions can be expressed for a pre-translation, and thus we require a more specific definition of a *translation*: Almost all components of the observational program calculi need to be translated, i.e. also a translation of contexts, generalized closing substitutions and convergence predicates is required. The advantages of these translations are: i) The required notion of convergence equivalence can be defined. ii) The notion of observational correctness can be introduced, which provides a systematic method to prove adequacy of the translation (see Proposition 3.16). iii) As argued in the introduction, if a context together with a convergence predicate is viewed as a specification of a program, then the more specific definition of a translation ensures that the specification itself can be translated.

3.2. Translations between Observational Program Calculi

In the following and in the remainder of the paper we consider *translations between observational program calculi*.

Definition 3.5. A *translation* $T : \mathcal{K} \to \mathcal{K}'$ between two observational program calculi $\mathcal{K} = (\mathcal{P}, \mathsf{Clos}, \mathcal{C}, \mathcal{O}, \mathcal{T}, \mathsf{type}, \mathcal{S})$

and $\mathcal{K}' = (\mathcal{P}', \mathsf{Clos}', \mathcal{C}', \mathcal{O}', \mathcal{T}', \mathsf{type}', \mathcal{S}')$ is a mapping from types to types $T : \mathcal{T} \to \mathcal{T}'$, programs to programs $T : \mathcal{P} \to \mathcal{P}'$, contexts to contexts $T : \mathcal{C} \to \mathcal{C}'$, and observation predicates to observation predicates $T : \mathcal{O} \to \mathcal{O}'$, such that:

1. $(p, \tau) \in \mathsf{type}$ implies $(T(p), T(\tau)) \in \mathsf{type}'$ and
   $(C, \tau_1, \tau_2)) \in \mathsf{type}$ implies $(T(C), T(\tau_1), T(\tau_2)) \in \mathsf{type}'$;

2. closedness is preserved for all p: $p \in \mathsf{Clos} \implies T(p) \in \mathsf{Clos}'$;

3. for all types τ, τ', contexts $C \in \mathcal{C}_{\tau,\tau'}$ and all $p \in \mathcal{P}_\tau$: $C(p) \in \mathsf{Clos} \implies T(C)(T(p)) \in \mathsf{Clos}'$; and

4. generalized closing substitutions are preserved, i.e., $T(\mathcal{S}) \subseteq \mathcal{S}'$. □

[2]Another approach to obtain full abstractness to ensure security in connection with compilation can be found in [4, 5], where a typed target language is used to ensure full abstractness. For the security issues this approach is reasonable, but for correctness of compilation one also has to consider target languages which are not typed.

Clearly, every such translation is also a pre-translation according to Definition 3.1 if it is restricted to types and programs only. The notion of translations is more restrictive as it must also apply to contexts, observation predicates, and even be compatible w.r.t. closedness with contexts as stated in condition 3. Conditions 2 and 4 are rather natural even though of technical nature. The relevance of condition 4 will become clear in Lemma 3.12. Conditions 2 and 3 are sufficiently relaxed to also permit for example dead code elimination that may turn non-closed (i.e. open) programs into closed ones.

Since convergence predicates can be translated arbitrarily, we can also compare observational program calculi with different numbers of convergence predicates. Given an observation predicate $\downarrow_{\Diamond}$ in $\mathcal{O}$, we will write $\downarrow_{\Diamond T}$ for its translation $T(\downarrow_{\Diamond})$ for convenience. We will also often use the notational convenience to indicate components in the image of a translation $T : \mathcal{K} \to \mathcal{K}'$ by a prime.

Lemma 3.6. *Let $T : \mathcal{K} \to \mathcal{K}'$ be a translation. Then $T(\mathcal{P}_\tau) \subseteq T(\mathcal{P})_{T(\tau)}$ and $T(\mathcal{C}_{\tau_1,\tau_2}) \subseteq T(\mathcal{C})_{T(\tau_1),T(\tau_2)}$*

Proof. This follows from Definition 3.5, condition (1). □

Lemma 3.7. *For all translations $T : \mathcal{K} \to \mathcal{K}'$, types τ_1, τ_2, programs p of type τ_1, and contexts C of type (τ_1, τ_2): If $C(p)$ is closed then both $T(C(p))$ and $T(C)(T(p))$ are closed.*

Proof. The first implication is condition (2) and the second implication is condition (3) of Definition 3.5 of translations. □

3.3. Observational Correctness

In this section we define additional properties of translations and emphasize which of these properties ensure "correctness" of the translation. Thereafter we will investigate relations between the different notions. We now consider the behavior of translations *operationally*, i.e. the following notion of *observational correctness* captures the intuition that compiled tests applied to compiled programs have the same result as in the source language, i.e. it ensures that the translation preserves and reflects the observations together with the contexts. This notion requires a translation on observational program calculi, and as we show gives a guide on how to prove adequacy of the translation.

Definition 3.8. A translation $T : \mathcal{K} \to \mathcal{K}'$ is called *observationally correct* (oc, for short) if it is convergence equivalent and compositional up to observations, where a translation T is:

convergence equivalent (ce) if for all $\downarrow_{\Diamond} \in \mathcal{O}$ and all closed p: $\left(p{\downarrow_{\Diamond}} \iff T(p){\downarrow_{\Diamond T}}\right)$, and

compositional up to observations (cuo) if the following condition holds: for all $\downarrow_{\Diamond} \in \mathcal{O}$, all types τ, τ', all contexts $C \in \mathcal{C}_{\tau,\tau'}$, and all programs $p \in \mathcal{P}_\tau$ such that $C(p)$ is closed: $T(C(p)){\downarrow_{\Diamond T}} \iff T(C)(T(p)){\downarrow_{\Diamond T}}$.

Example 3.9 (Example 3.3, cont.)**.** *We can extend the pre-translation T_{incl} from PCF_{cbv} into $\mathrm{PCF}_{cbv,\oplus}$ to a translation, by defining: $T_{incl}(C) = C$ and $T_{incl}(\downarrow_{\mathrm{PCF}\,cbv}) = \downarrow_{\mathrm{PCF}\,cbv}$. Obviously, this translation is convergence equivalent and compositional, so it is observationally correct.*

Also the pre-translation $T_{-\mathbb{B}} :: \mathrm{PCF}_{cbv} \to \mathrm{PCF}_{cbv,-\mathbb{B}}$ can easily be extended to contexts, which are translated like expressions and $T([.]_\tau) = [.]_{T(\tau)}$, and convergence is mapped to convergence. One can also prove that all other conditions on translations are satisfied. The translation $T_{-\mathbb{B}}$ is compositional and thus also (cuo)*. For proving convergence equivalence, we first observe that values are translated into values, i.e. $p \in \mathrm{PCF}_{cbv}$ is a value iff $T(p) \in \mathrm{PCF}_{cbv,-\mathbb{B}}$ is a value. Also for any program p and context C one can observe that $T(C(p)) = T(C)(T(p))$ such that $T(C)$ is a reduction context of $\mathrm{PCF}_{cbv,-\mathbb{B}}$ iff C is a reduction context of PCF_{cbv}. All reduction steps directly correspond, i.e. $p \to p'$ iff $T(p) \to T(p')$ except for reducing the constant* **zero**? *where one reduction is replaced by two reduction, i.e. $R(\mathbf{zero}?\ i) \to R(b)$ iff $T(R(\mathbf{zero}?\ i)) \to p' \to T(R(b))$. Hence, by induction on the length of converging reduction sequences of p ($T(p)$ resp.) convergence equivalence can be proved. Thus $T_{-\mathbb{B}}$ is observationally correct.*

Note that convergence equivalence is often called "computational adequacy" when relating an operational semantics with a denotational semantics. However, observational correctness additionally requires (a weak form of) compositionality, which allows to translate contexts separately from programs. Of course compositionality up to observations is a generalization of usual compositionality and its variants:

Definition 3.10. Let $T : \mathcal{K} \to \mathcal{K}'$ be a translation. Then

1. T is *compositional* iff for all $\tau, \tau' \in \mathcal{T}$, for all $C \in \mathcal{C}_{\tau,\tau'}$ and $p \in \mathcal{P}_\tau$, if $C(p)$ is closed then $T(C(p)) = T(C)(T(p))$.

2. T is *compositional modulo* $\sim$, iff for all types τ_1, τ_2, $p \in \mathcal{P}_{\tau_1}$, $C \in \mathcal{C}_{\tau_1,\tau_2}$: if $C(p)$ is closed, then $T(C(p)) \sim'_{T(\tau_2)} T(C)(T(p))$.

Lemma 3.11. *Let $T : \mathcal{K} \to \mathcal{K}'$ be a translation. Then T is (*cuo*) provided it is compositional or compositional modulo $\sim$.*

Proof. If T is compositional, then (cuo) holds obviously. Let T be compositional modulo $\sim$ and assume that $C(p)$ is closed. Then Lemma 3.7 implies that $T(C(p))$ and $T(C)(T(p))$ are closed. Lemma 2.7 is now applicable and shows that for all $\downarrow$: $T(C(p))\downarrow_T \iff T(C)(T(p))\downarrow_T$. □

Observational correctness has a more explicit description by a homomorphism-like condition: The translation retains the results of applying contexts and then applying a convergence test.

Lemma 3.12 (Alternative characterization of (oc))**.** *Suppose $T : \mathcal{K} \to \mathcal{K}'$ is a translation. Then T is observationally correct if, and only if:*

(acooc) *For all $\tau, \tau' \in \mathcal{T}$, all $p \in \mathcal{P}_{\tau_1}$, $C \in \mathcal{C}_{\tau_1,\tau_2}$:*
if $C(p)$ is closed, then for all $\downarrow \in \mathcal{O}$: $C(p)\downarrow \iff T(C)(T(p))\downarrow_T$.

Proof. (oc) $\Longrightarrow$ (acooc): Let T be (oc), i.e., convergence equivalent and compositional up to observations; let C be a context and p be a program such that $C(p)$ is closed. Then $C(p)\downarrow_\Downarrow \overset{\mathsf{ce}}{\Longleftrightarrow} T(C(p))\downarrow_{\Downarrow_T} \overset{\mathsf{cuo}}{\Longleftrightarrow} T(C)(T(p))\downarrow_{\Downarrow_T}$ and thus the claim holds.

(acooc) $\Longrightarrow$ (oc): Let p be a closed program. According to Definition 2.4 there is a $C \in \mathcal{S}$, such that $C(p)$ is also closed and $C(p)\downarrow_\Downarrow \iff p\downarrow_\Downarrow$. Since $T(C) \in \mathcal{S}'$ by condition (4) of Definition 3.5 and $T(p)$ is closed, the following equivalence holds: $p\downarrow_\Downarrow \iff C(p)\downarrow_\Downarrow \overset{\mathsf{acooc}}{\Longleftrightarrow} T(C)(T(p))\downarrow_{\Downarrow_T} \iff T(p)\downarrow_{\Downarrow_T}$. Hence T is (ce). It remains to show that T is (cuo): Assume $C(p)$ is closed, then $T(C(p))\downarrow_{\Downarrow_T} \overset{\mathsf{ce}}{\Longleftrightarrow} C(p)\downarrow_\Downarrow \overset{\mathsf{acooc}}{\Longleftrightarrow} T(C)(T(p))\downarrow_{\Downarrow_T}$. □

Let us also emphasize that Definitions 3.2, 3.5 and 3.8 are stated only in terms of observational program calculi, and hence they can be used for all calculi with such a description.

In the remainder of this section, we show differences and similarities between the introduced properties of translations, and we give some small examples for translations.

3.4. Comparing the Properties of Translations

In this section we compare the different correctness notions and show how they are related. First, Proposition 3.4 can be extended to translations, convergence equivalence, and observational correctness: the introduced correctness properties are closed under the composition of translations:

Proposition 3.13 (Closure under composition)**.** *Let $\mathcal{K}, \mathcal{K}', \mathcal{K}''$ be program calculi, and $T : \mathcal{K} \to \mathcal{K}'$, $T' : \mathcal{K}' \to \mathcal{K}''$ be translations. Then $T' \circ T : \mathcal{K} \to \mathcal{K}''$ is also a translation. For every property P from Definitions 3.2, for the properties (ce) and (oc) from Definition 3.8, and for the property "compositional" from Definition 3.10: if T and T' have property P, then so has their composition $T' \circ T$.*

Proof. For adequacy, full abstractness, and the isomorphism property the claim follows from Proposition 3.4, since every translation is also a pre-translation. If T and T' are compositional, then clearly also $T' \circ T$ is compositional. We consider the remaining properties:

(ce): Let $\downarrow \in \mathcal{O}$ and p be closed. Then $p\downarrow_\Downarrow \iff T(p)\downarrow_{\Downarrow_T}$, since T is (ce), also $T(p)$ is closed and thus $T(p)\downarrow_{\Downarrow_T} \iff T'(T(p))\downarrow_{\Downarrow_{T'\circ T}}$, since T' is (ce). In summary, this shows $p\downarrow_\Downarrow \iff (T' \circ T)(p)\downarrow_{\Downarrow_{T'\circ T}}$.

(oc): Let T, T' be observationally correct. Since this implies that T and T' are also convergence equivalent, the previous item already shows that $T' \circ T$ is (ce). It remains to show that $T' \circ T$ is (cuo). Let $C(p)$ be closed. Then $T(C(p))\downarrow_{\Downarrow_T} \iff T(C)(T(p))\downarrow_{\Downarrow_T}$ holds, since T is (cuo). Since $T(C)(T(p))$ is closed and T' is (cuo), we also have $T'(T(C)(T(p)))\downarrow_{\Downarrow_{T'\circ T}} \iff T'(T(C))(T'(T(p)))\downarrow_{\Downarrow_{T'\circ T}}$. This shows $T(C(p))\downarrow_{\Downarrow_T} \iff T'(T(C))(T'(T(p)))\downarrow_{\Downarrow_{T'\circ T}}$. Since T' is convergence equivalent, we also have $T(C(p))\downarrow_{\Downarrow_T} \iff T'(T(C(p)))\downarrow_{\Downarrow_{T'\circ T}}$ which implies $T'(T(C(p)))\downarrow_{\Downarrow_{T'\circ T}} \iff T'(T(C))(T'(T(p)))\downarrow_{\Downarrow_{T'\circ T}}$, and thus $T' \circ T$ is (cuo). □

Remark 3.14. We show that the previous Proposition 3.13 does not hold for (cuo). As a counter-example let $\mathcal{K}_i = (\mathcal{P}_i, \mathsf{Clos}_i, \mathcal{C}_i, \{\downarrow_{\forall i}\}, \mathcal{T}, \mathsf{type}, \mathcal{S})$ for $i = 1, 2, 3$ where $\mathsf{Clos}_i = \mathcal{P}_i$ (all programs are closed), $\mathcal{T} = \{\mathsf{unit}\}$ (a single type), $\mathsf{type} = \{(p, \mathsf{unit}) \mid p \in \mathcal{P}_i\} \cup \{C, (\mathsf{unit}, \mathsf{unit}) \mid C \in \mathcal{C}\}, \mathcal{S} = \{[.]\}$ (all programs and contexts are typeable in the obvious way). We set $\mathcal{P}_1 = \{a, b\}$, $\mathcal{C}_1 = \{[.], C\}$ with $C(a) = a = C(b)$, and $\downarrow_{\forall 1} = \{a, b\}$; $\mathcal{P}_2 = \{A, B\}$, $\mathcal{C}_2 = \{[.]\}$, and $\downarrow_{\forall 2} = \emptyset$; and $\mathcal{P}_3 = \{\alpha, \beta\}$, $\mathcal{C}_3 = \{[\cdot]\}$ and $\downarrow_{\forall 3} = \{\alpha\}$. Let $T_{1,2} : \mathcal{K}_1 \to \mathcal{K}_2$ be the translation with $T_{1,2}(a) = A, T_{1,2}(b) = B, T_{1,2}(C) = [.]$, $T_{1,2}(\mathsf{unit}) = \mathsf{unit}$, and let $T_{2,3} : \mathcal{K}_2 \to \mathcal{K}_2$ be the translation with $T_{2,3}(A) = \alpha$, $T_{2,3}(B) = \beta$ and $T_{2,3}([.]) = [.]$. Then $T_{1,2}$ clearly is (cuo) (since all programs of $\mathcal{P}_2$ diverge). Also $T_{2,3}$ is (cuo), since $T_{2,3}([A]) = T_{2,3}([.])(T_{2,3}(A)) = \alpha$ and $T_{2,3}([B]) = T_{2,3}([.])(T_{2,3}(B)) = \beta$.

However, the composition $T_{2,3} \circ T_{1,2}$ is not (cuo): $T_{2,3}(T_{1,2}(C(b)))\downarrow_{\forall 3}$, since $T_{2,3}(T_{1,2}(a)) = T_{2,3}(A) = \alpha$, but $\neg(T_{2,3}(T_{1,2}(C))(T_{2,3}(T_{1,2}(b)))\downarrow_{\forall 3})$, since $T_{2,3}(T_{1,2}(C))(T_{2,3}(T_{1,2}(b))) = T_{2,3}([.])(T_{2,3}(T_{1,2}(b))) = T_{2,3}(T_{1,2}(b)) = T_{2,3}(B) = \beta$.

For compositionality modulo $\sim$ and cuo an extra requirement is necessary:

Proposition 3.15. *Let T, T' be translations such that T' is fully abstract. If T and T' are compositional modulo $\sim$, then their composition $T' \circ T$ is compositional modulo $\sim$. If T and T' are* cuo*, then their composition $T' \circ T$ is* cuo*.*

Proof. We have to show that $(T' \circ T)(C(p)) \sim'' T'(T(C))(T'(T(p)))$ for closed $C(p)$. The translation T is compositional modulo $\sim$, hence $T(C(p)) \sim' T(C)(T(p))$, and since T' is compositional modulo $\sim$, also $T'(T(C)(T(p))) \sim'' T'(T(C))(T'(T(p)))$. Since T' is fully abstract, $T(C(p)) \sim' T(C)(T(p))$ implies $T'T(C(p)) \sim'' T'(T(C)(T(p)))$, hence the first claim holds. The second claim follows from the first and Lemma 3.11. □

The following result links the operational correctness notions and the correctness notions concerning the preorders of the source and the target calculus:

Theorem 3.16. *If translation $T : \mathcal{K} \to \mathcal{K}'$ is observationally correct, then T is also adequate.*

Proof. To show adequacy, let us assume that $T(p_1) \leq'_{T(\tau)} T(p_2)$. We must prove that $p_1 \leq_\tau p_2$. Thus let C be such that $C(p_1)$ and $C(p_2)$ are closed and $C(p_1)\downarrow$. By the characterization of observational correctness in Lemma 3.12, this implies $T(C)(T(p_1))\downarrow_{\forall T}$, where $T(C)(T(p_1))$ and $T(C)(T(p_2))$ are closed by Lemma 3.7. From $T(p_1) \leq'_{T(\tau)} T(p_2)$, we obtain $T(C)(T(p_2))\downarrow_{\forall T}$. The other direction of the equivalence in the observational correctness condition implies $C(p_2)\downarrow$. This proves $p_1 \leq_\tau p_2$. □

This result also shows that observational correctness provides *a method* on how to prove adequacy of a translation. It is sufficient to show convergence equivalence provided the translation is compositional (or even only compositional up to observation). The value of this approach is that the proof of convergence equivalence does not require to reason about all contexts which is usually hard. So observational correctness separates the contextual reasoning from the reasoning about convergence.

Note that several papers (e.g. [57, 5]) use a similar pattern for proving adequacy (or sometimes called "equivalence reflection") of (almost) compositional translations. We also gave an abstract proof for a class of observational program calculi in [60].

Example 3.17. *For our running examples (Example 3.9), observational correctness of T_{incl} and $T_{-\mathbb{B}}$ and Proposition 3.16 imply that both translations are adequate.*

The following proposition shows: (ce) is in general not sufficient for adequacy, and full abstractness is not implied by observational correctness. Similarly, (ce) is not even implied by full abstractness (and thus neither by adequacy).

Proposition 3.18. *The following holds:*

1. *Convergence equivalence does not imply adequacy.*
2. *Observational correctness does not imply full abstractness.*
3. *Convergence equivalence is not implied by the conjunction of compositionality up to observations and preservation and reflection of $\leq$.*
4. *Convergence equivalence and full abstractness do not imply observational correctness.*

Proof. 1. Let the observational program calculus $\mathcal{K}$ have three programs: a, b, c with $a{\downarrow_{\mathcal{K}}}$, $b{\downarrow_{\mathcal{K}}}$ and $c{\downarrow_{\mathcal{K}}}$. Assume there are contexts C_1, C_2 with $C_1 = Id$ and $C_2(a) = a$, $C_2(b) = a$, $C_2(c) = c$. Then $b \not\sim_{\mathcal{K}} c$. The language $\mathcal{K}'$ has three programs A, B, C with $A{\downarrow_{\mathcal{K}'}}$, $B{\downarrow_{\mathcal{K}'}}$ and $C{\downarrow_{\mathcal{K}'}}$. There is only the identity context Id' in $\mathcal{K}'$. Then $B \sim_{\mathcal{K}'} C$. Let the translation be defined as $T : \mathcal{K} \to \mathcal{K}'$ with $T(a) = A$, $T(b) = B$, $T(c) = C$, $T(C_1) = T(C_2) = Id'$, and $T({\downarrow_{\mathcal{K}}}) = {\downarrow_{\mathcal{K}'}}$. Then T is convergence equivalent, but neither adequate nor observationally correct. T is also not (cuo), since $T(C_2(b)) = A$ and thus $T(C_2(b)){\uparrow_{\mathcal{K}'}}$ while $T(C_2)(T(b)) = Id(B) = B$ and thus $T(C_2)(T(b)){\downarrow_{\mathcal{K}'}}$.

2. Example 3.25 is a witness for this. Another simple example taken from [34] is the identity encoding into (call-by-name) PCF with product types from PCF but without the projections **fst** and **snd**. Then, in the restricted calculus, all pairs are indistinguishable but the presence of the contexts $\mathbf{fst}\,[.]_{(\tau,\tau')}$ and $\mathbf{snd}\,[.]_{(\tau,\tau')}$ in PCF with products permits more distinctions to be made.

3. A trivial example is given by two calculi $\mathcal{K}$ with $p{\downarrow_{\mathcal{K}}}$ for all p, and $\mathcal{K}'$ with the same programs and $p{\uparrow_{\mathcal{K}'}}$ for all p. For the identity translation $T(p) = p, T({\downarrow_{\mathcal{K}}}) = {\downarrow_{\mathcal{K}'}}$ for all p it is clear that $\forall p_1, p_2 : p_1 \leq p_2 \iff T(p_1) \leq' T(p_2)$ holds, and that the translation is compositional up to observations, but clearly T does not preserve convergence.

4. Let $\mathcal{K}$ have two programs a, b, the identity context, and one context C with $C(a) = b$ and $a{\downarrow_{\mathcal{K}}}$, $b{\uparrow_{\mathcal{K}}}$. Let $\mathcal{K}'$ consists of A, B with $A{\downarrow_{\mathcal{K}'}}$, and $B{\uparrow_{\mathcal{K}'}}$, and the identity context. Let $T : \mathcal{K} \to \mathcal{K}'$ be the translation defined by $T(a) = A, T(b) = B$, $T(C) = Id'$, and $T({\downarrow_{\mathcal{K}}}) = {\downarrow_{\mathcal{K}'}}$. Then T is $\leq$-preserving and reflecting, since there are no relevant equalities. It is also convergence equivalent. But it is not observationally correct, since $T(C(a)) = T(b) = B$, i.e. $T(C(a)){\uparrow_{\mathcal{K}'}}$, and $T(C)(T(a)) = A$, i.e. $T(C)(T(A)){\downarrow_{\mathcal{K}'}}$. □

Our definition of translations implies that the image of a translation is also an observational program calculus: Consider a translation $T : \mathcal{K} \to \mathcal{K}'$, where $\mathcal{K} = (\mathcal{P}, \mathsf{Clos}, \mathcal{C}, \mathcal{O}, \mathcal{T}, \mathsf{type}, \mathcal{S})$ and $\mathcal{K}' = (\mathcal{P}', \mathsf{Clos}', \mathcal{C}', \mathcal{O}', \mathcal{T}', \mathsf{type}', \mathcal{S}')$ then the image of $\mathcal{K}$ under T, denoted $T(\mathcal{K})$, is also an observational program calculus $T(\mathcal{K}) := (T(\mathcal{P}), T(\mathsf{Clos}), T(\mathcal{C}), T(\mathcal{O}), T(\mathcal{T}), T(\mathsf{type}), T(\mathcal{S}))$ where $T(\mathsf{type})$ is the image of type as a relation.

Note that this definition is only possible, since contexts and programs are translated separately by T, and by the conditions in the definition of translations. We will use the symbol $\leq''_\tau$ for the (type-indexed) observational preorder of $\mathcal{K}'' = T(\mathcal{K})$.

A further preorder for the image calculus $T(\mathcal{K})$ can be defined, which only allows to compare programs with the same source types, and also only uses contexts which are applicable for source programs.

We call this preorder $\leq_{T,\tau}$, which is defined on $\mathcal{K}'' = T(\mathcal{K})$ but with $\mathcal{K}$-type τ. It is defined as follows: let τ be a $\mathcal{K}$-type. For programs $p'_1, p'_2 \in \mathcal{P}'_{T(\tau)}$ the inequation $p'_1 \leq_{T,\tau} p'_2$ holds, iff for all $\downarrow_T \in T(\mathcal{O})$, all $p_1, p_2 \in \mathcal{P}_\tau$ with $T(p_1) = p'_1, T(p_2) = p'_2$, and all $C \in \mathcal{C}_{\tau,\tau'}$, such that $T(C)(T(p_i))$ are closed for $i = 1, 2$: $T(C)(T(p_1))\downarrow_T \implies T(C)(T(p_2))\downarrow_T$.

We will now show that observing the translated programs using translated contexts under type restrictions makes the same distinctions between the original and the translated programs if the translation is observationally correct and in addition *closedness reflecting*[3]:

Definition 3.19. A translation T is *closedness reflecting* iff for all programs p and contexts C: $C(p)$ is closed $\iff$ $T(C)(T(p))$ is closed.

Lemma 3.20. *Let $T : \mathcal{K} \to \mathcal{K}'$ and let T be closedness reflecting. Then $T(C(p))$ is closed provided $C(p)$ is closed or $T(C)(T(p))$ is closed.*

Theorem 3.21. *Let $\mathcal{K}, \mathcal{K}'$ be observational program calculi and $T : \mathcal{K} \to \mathcal{K}'$ be an observationally correct and closedness reflecting translation. Let $\mathcal{K}'' := T(\mathcal{K})$ be the image subcalculus of $\mathcal{K}'$ under T and for $\tau' \in \mathcal{T}'$, let $\leq''_{\tau'}$ be the observational preorder of $\mathcal{K}''$, and for $\tau \in \mathcal{T}$, let $\leq_{T,\tau}$ be the preorder on $\mathcal{K}''$ (on programs of types $T(\tau)$) as defined above. Then the following holds:*

1. *For all types $\tau \in \mathcal{T}$ and programs $p_1, p_2 \in \mathcal{P}_\tau$: $p_1 \leq_\tau p_2 \iff T(p_1) \leq_{T,\tau} T(p_2)$.*
2. *If T is injective on $\mathcal{T}$, then for all types τ and programs $p_1, p_2 \in \mathcal{P}_\tau$: $p_1 \leq_\tau p_2 \iff T(p_1) \leq''_{T(\tau)} T(p_2)$, i.e. the restricted translation $T : \mathcal{K} \to \mathcal{K}''$ is fully-abstract and also an isomorphism between $\mathcal{K}$ and $\mathcal{K}''$*

Proof. (1) Let $T(p_1) \leq_{T,\tau} T(p_2)$ hold, and let $C \in \mathcal{C}_{\tau,\tau'}$ such that $C(p_1)$, $C(p_2)$ are closed and $C(p_1)\downarrow$. Then also $T(C)(T(p_1))$ and $T(C)(T(p_2))$ are closed (by Lemma 3.7). Observational correctness implies $T(C)(T(p_1))\downarrow_T$. The definition of $\leq_{T,\tau}$ implies $T(C)(T(p_2))\downarrow_T$ and by observational correctness, we obtain $C(p_2)\downarrow$. Thus $p_1 \leq_\tau p_2$.

[3] Note that closedness reflection cannot be derived from observational correctness (see Remark 4.10).

Let $p_1 \leq_\tau p_2$ and let $C \in \mathcal{C}_{\tau,\tau'}$ such that $T(C)(T(p_1))$, $T(C)(T(p_2))$ are closed, and let $T(C)(T(p_1))\downarrow_{T}$. We show that $T(C)(T(p_2))\downarrow_{T}$. By closedness reflection of T, $C(p_1)$ and $C(p_2)$ are closed. Observational correctness, the characterization in Lemma 3.12, $p_1 \leq_\tau p_2$ and Lemma 2.7 show $T(C)(T(p_1))\downarrow_{T} \implies C(p_1)\downarrow \text{ and } C(p_1)\downarrow \implies C(p_2)\downarrow$. Then $C(p_2)\downarrow$ in turn implies $T(C)(T(p_2))\downarrow_{T}$. Since this holds for all contexts $T(C)$ satisfying the conditions, we have shown $T(p_1) \leq_{T,\tau} T(p_2)$.

(2) Adequacy follows from Proposition 3.16 by applying it to the restricted translation $T : \mathcal{K} \to \mathcal{K}''$. Let $p_1 \leq_\tau p_2$ and let C' be a $\mathcal{K}''$-context such that $C'(T(p_1))$, $C'(T(p_2))$ are closed and $C'(T(p_1))\downarrow_{T}$. We show that $C'(T(p_2))\downarrow_{T}$. Since T is injective on $\mathcal{T}$, the translation $T : \mathcal{K} \to \mathcal{K}''$ is surjective from $\mathcal{C}_{\tau,\tau'} \to \mathcal{C}''_{T(\tau),T(\tau')}$: If a triple $(C', \tau_1', \tau_2') \in T(\mathsf{type})$, then there is a triple $(C, \tau_1, \tau_2) \in \mathsf{type}$, with $T(\tau_1) = \tau_1'$ and $T(\tau_2) = \tau_2'$. Injectivity on types implies that τ_1, τ_2 are unique, hence surjectivity holds. Thus there is a context C with input type τ, such that $T(C) = C'$. Clearly, since T is closedness reflecting, $C(p_1)$ and $C(p_2)$ are closed. Observational correctness and Lemma 3.12 show $C'(T(p_1))\downarrow_{T} \Leftrightarrow T(C)(T(p_1))\downarrow_{T}$ and $T(C)(T(p_1))\downarrow_{T} \implies C(p_1)\downarrow$ and $C(p_1)\downarrow \implies C(p_2)\downarrow$. This in turn implies $T(C)(T(p_2))\downarrow_{T}$ which is equivalent to $C'(T(p_2))\downarrow_{T}$. Since this holds for all contexts C', we have shown $T(p_1) \leq''_{T(\tau)} T(p_2)$. □

Corollary 3.22. *Let $\mathcal{K}, \mathcal{K}'$ be observational program calculi and $T : \mathcal{K} \to \mathcal{K}'$ be an observationally correct and closedness reflecting translation, such that T is an isomorphism on the type structure, surjective on programs, contexts, and observation predicates. Then $T : \mathcal{K} \to \mathcal{K}'$ is an isomorphism.*

Proof. This follows from Theorem 3.21 (2). □

For many translations, being injective on types is too strict a requirement, for instance, if a newly added data structure is compiled into existing ones. However, in these cases, provided the translation is observationally correct and closedness reflecting, part (1) of Theorem 3.21 is applicable (see also Remark 4.12 and Example 4.13).

Example 3.23. *The classical Church-encoding of the untyped lambda calculus with pairs and selectors into the pure untyped lambda calculus is neither convergence equivalent nor adequate. The problem is that dynamically untyped programs like* **fst** $(\lambda x.x)$ *are translated into convergent programs. An observationally correct (and thus also adequate) encoding can be established if the lambda calculus with pairs is simply typed (and thus the above mentioned counter-examples are excluded) and the lambda calculus without pairs is untyped. In Section 5.2 the encodings and proofs are given in detail.*

3.5. Further Examples

We provide some simple uses of translations which illustrate the introduced properties.

Example 3.24. *Let* $\mathrm{PCF}_{cbv,-\mathbf{fix}}$ *be the restriction of* PCF_{cbv} *from Section 2.1 where the fixed point operators are dropped. The embedding $\iota : \mathrm{PCF}_{-\mathbf{fix}} \to \mathrm{PCF}_{cbv}$ is defined as the identity on types and expressions, and hence ι is compositional and (*ce*), hence adequate by Proposition 3.16. Below in Example 4.4 we show that it is also fully abstract.*

Example 3.25. *We give an example of an adequate, but not fully abstract, embedding: Let* $\text{PCF}_{cbv,-\mathbf{fix},0}$ *be* $\text{PCF}_{cbv,-\mathbf{fix}}$ *(see Example 3.24) with the additional reduction rule* $\mathbf{pred}\ 0 \to 0$ *and let* $\text{PCF}_{cbv,0}$ *be* PCF_{cbv} *with the same modification. The embedding* $\iota : \text{PCF}_{cbv,-\mathbf{fix},0} \to \text{PCF}_{cbv,0}$ *is the identity on types and expressions, and hence the embedding is compositional and (*ce*), hence adequate. However, the embedding is not fully abstract: The expressions* $p_1 = \lambda x.0$ *and* $p_2 = \lambda x.\mathbf{if}\ (\mathbf{zero}?\ (x\ 0))\ \mathbf{then}\ 0\ \mathbf{else}\ 0$ *are observationally equivalent w.r.t.* $\text{PCF}_{cbv,-\mathbf{fix},0}$*, since* $\text{PCF}_{cbv,-\mathbf{fix},0}$ *is a simply typed lambda calculus where every closed expression is terminating, but are not observationally equivalent w.r.t.* $\text{PCF}_{cbv,0}$*: They can be distinguished by applying both to* $\lambda y.\bot$*, where* $\bot$ *is a non-converging* $\text{PCF}_{cbv,0}$*-expression.*

Finally, we compare call-by-name PCF and call-by-value PCF.

Example 3.26. *Let* PCF_{cbn} *be call-by-name* PCF *and* PCF_{cbv} *be call-by-value* PCF *and let* $T : \text{PCF}_{cbn} \to \text{PCF}_{cbv}$ *be given by the identity on types, programs, and contexts. Then* T *is compositional, but it is not (*ce*) since, for example, the expression* $(\lambda x.0)\ \bot$ *has different convergence behaviors in call-by-name and call-by-value* PCF*. Since* $(\lambda x.0)\ \bot \sim 0$ *in* PCF_{cbn} *but not in* PCF_{cbv}*, the translation is not fully abstract. In the converse direction, the expressions* $\lambda x.0$ *and* $\lambda x.\big(\mathbf{if}\ x\ \mathbf{then}\ 0\ \mathbf{else}\ 0\big)$ *are equivalent in* PCF_{cbv}*, but not in* PCF_{cbn}*, hence the translation is also not adequate.*

4. Obtaining Full Abstractness for Language Extensions

We now consider cases and techniques where besides observational correctness and adequacy also full abstractness or even the isomorphism property of a translation can be shown. We only consider the case of language extensions or, taking a slightly more general point of view, embeddings of one language into another, which generalizes the situation of Corollary 3.22 in several respects. A typical case is that new language primitives are added to a calculus, together with their (operational) semantics, which is represented by an *encoding*, which is also a translation.

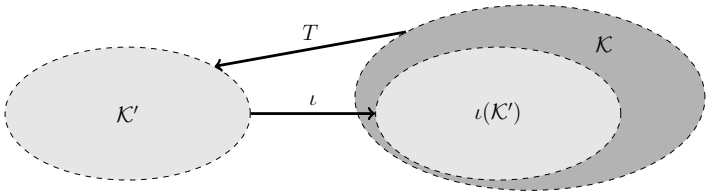
One issue is whether the extension is conservative, i.e. whether the embedding of the non-extended language into the extended one is fully abstract. This ensures that the newly added primitives cannot be used for distinguishing existing programs. The second issue is whether the encoding of the extended language into the base language is fully abstract. If also the embedding is fully abstract, then the extension can be viewed as 'syntactic sugar'.

4.1. Extensions and Embeddings

Definition 4.1 (Extension and embedding). An observational program calculus $\mathcal{K}$ is an *extension* of the observational program calculus $\mathcal{K}'$ iff there is a translation $\iota : \mathcal{K}' \to \mathcal{K}$ which is observationally correct, i.e. (ce) and (cuo), and which is injective on the types. In this case the translation ι is called an *embedding*. If the embedding is in addition fully abstract, then it is also called a *conservative embedding*.

Note that an embedding ι is adequate by Proposition 3.16, i.e. injective modulo $\sim$, but in general not necessarily fully abstract. Informally, the embedding situation can be described

(after identifying $\mathcal{K}'$-programs with their image under ι and modulo $\sim$) as follows: every $\mathcal{K}'$-type is also a $\mathcal{K}$-type, $\mathcal{P}'_\tau \subseteq \mathcal{P}_\tau$, and $\mathcal{C}'_{\tau,\tau'}$ can be seen as a subset of $\mathcal{C}_{\tau,\tau'}$, more precisely: $\mathcal{C}'_{\tau,\tau'}$ is the same as $\{C\restriction_{\iota(\mathcal{P}')} \mid C \in \mathcal{C}_{\tau,\tau'}\}$, (where $f\restriction_A$ is the function f restricted to the set A) and the observation predicates coincide on $\mathcal{K}'$-programs.



If $\mathcal{K}$ is an extension of $\mathcal{K}'$, then an observationally correct translation $T : \mathcal{K} \to \mathcal{K}'$ (the encoding), where $T \circ \iota$ is the identity, has the nice consequence of T and ι being fully abstract under certain preconditions (see Theorem 4.6, Corollaries 4.7 and 4.8; and see also Corollary 3.22).

4.2. Conservativeness of Embeddings

We first look for the issue whether $\mathcal{K}$ extends $\mathcal{K}'$ conservatively, i.e. whether ι is a conservative embedding, or in other words; whether ι is fully abstract. This includes the case that there is no sensible back-translation (the encoding) $T : \mathcal{K} \to \mathcal{K}'$. If full abstractness of ι can be shown, then all equations that hold in $\mathcal{K}'$ also hold for $\mathcal{K}$ and thus the newly added language constructs cannot be used to distinguish the "old" programs. This property is of practical importance, since e.g. optimizations that are valid for $\mathcal{K}'$ are still valid for $\mathcal{K}$ (restricted to $\mathcal{K}'$) programs. On the other hand, disproving conservativity implies that the new constructs add expressivity to the language.

A classical and paradigmatic example is the extension of the call-by-name lambda calculus (or call-by-name PCF) by a "parallel or", and the question whether this extension changes the semantics. The classical answer is "yes": the semantics is changed [2]: a parallel-or construct **por** added to the call-by-name lambda calculus (call-by-name PCF) permits to observationally distinguish two functions, which were contextually equal without the extension. Therefore consider the function F (of type $o \to (o \to o \to o) \to o$), where `not` and `and3` are defined as usual:

$$F := \lambda x.\lambda f.\ \textbf{if}\ \texttt{and3}\ (f\ \textbf{true}\ \bot)\ (f\ \bot\ \textbf{true})\ (\texttt{not}\ (f\ \textbf{false}\ \textbf{false}))\ \textbf{then}\ x\ \textbf{else true}$$

Then $(F\ \textbf{true}) \sim (F\ \textbf{false})$ in PCF_{cbn} but $(F\ \textbf{true}) \not\sim (F\ \textbf{false})$ in the extension of PCF_{cbn} with **por** since $(F\ \textbf{true}\ \textbf{por}) \sim \textbf{true}$ and $(F\ \textbf{false}\ \textbf{por}) \sim \textbf{false}$, but $\textbf{true} \not\sim \textbf{false}$.

There are other investigations into extensions and whether these are conservative. For example in [16] and for more calculi in [58], among others, the question whether the lazy lambda calculus [1] is conservatively extended by a `strict` operator was answered negatively. We will discuss larger examples for these conservativity questions in Section 5.3.

We provide a criterion for full abstractness of the embedding in the case where the extended calculus cannot be translated into the base calculus. The main idea is to use a *family* of translations as an approximation of a translation that cannot be otherwise

represented. For example, if the translation has to deal with recursive programs (or types), then it may be possible to consider (all) the finitary approximations instead.

The following definition intuitively describes that two programs have the same sets of free variables. However, our framework has no notion of free variables, and thus the definition is given in terms of contexts and the closedness property:

Definition 4.2. For $p, p' \in \mathcal{P}_\tau$ we say p and p' are *contextually closedness equivalent*, denoted by $p \cong_{cl} p'$, iff for all contexts C of type (τ, τ'): $C(p)$ is closed iff $C(p')$ is closed.

Theorem 4.3 (Families of Translations)**.** *Let $\mathcal{K}$ be an extension of $\mathcal{K}'$, i.e. with an observationally correct translation $\iota : \mathcal{K}' \to \mathcal{K}$. Let J be a partially ordered index set, such that for all j_1, j_2, there is some $j_3 \in J$ with $j_1 \leq j_3$ and $j_2 \leq j_3$. Let $\{T_j\}_j$ be a J-indexed family of translations $T_j : \mathcal{K} \to \mathcal{K}', j \in J$ such that the following conditions hold:*

1. *For all $j \in J$: $T_j \circ \iota$ is the identity on $\mathcal{O}'$ and on $\mathcal{K}'$-types.*
2. *for all $j \in J$ and for all types τ': $(T_j \circ \iota)(p') \sim'_{\tau'} p'$ for all $\mathcal{K}'$-programs p' of type τ';*
3. *for all $j \in J$ and for all $\mathcal{K}'$-programs p': $(T_j \circ \iota)(p') \cong_{cl} p'$.*
4. *for every context $C \in \mathcal{C}_{\tau,\tau'}$ and program $p \in \mathcal{P}_\tau$ where $C(p)$ is closed, there is some $k \in J$, such that for all $j \in J$ with $j \geq k$ and for all $\downarrow$ in $\mathcal{O}$: $C(p)\downarrow \Leftrightarrow T_j(C)(T_j(p))\downarrow_T$.*

Then ι is fully abstract, i.e. ι is a conservative embedding.

Proof. Proposition 3.16 implies that ι is adequate. In order to show full abstractness of ι, let $p'_1, p'_2 \in \mathcal{P}'$ be programs of type τ and $\downarrow' \in \mathcal{O}'$ an observation predicate such that $p'_1 \leq'_{\downarrow',\tau} p'_2$ and let C be an arbitrary $\mathcal{K}$-context such that $C(\iota(p'_1))$, $C(\iota(p'_2))$ are closed, and $C(\iota(p'_1))\downarrow_\iota$. For contradiction, let us assume that $C(\iota(p'_2))\uparrow_\iota$. Then there is some k_1 such that for all $j_1 \geq k_1$: $T_{j_1}(C)(T_{j_1}(\iota(p'_1)))$ is closed and $T_{j_1}(C)(T_{j_1}(\iota(p'_1)))\downarrow'$. There is also some k_2 such that for all $j_2 \geq k_2$: $T_{j_2}(C)(T_{j_2}(\iota(p'_2)))$ is closed but $T_{j_2}(C)(T_{j_2}(\iota(p'_2)))\uparrow'$. Due to the condition on the order, there is some common $k_3 \in J$ with $k_1 \leq k_3$, $k_2 \leq k_3$ such that for all $j \geq k_3$: $T_j(C)(T_j(\iota(p'_1)))$ and $T_j(C)(T_j(\iota(p'_2)))$ are closed, and $T_j(C)(T_j(\iota(p'_1)))\downarrow'$, but $T_j(C)(T_j(\iota(p'_2)))\uparrow'$.

By condition (3), $T_j(C)(p'_1)$ as well as $T_j(C)(p'_2)$ are closed, and by condition (2), we obtain $T_j(C)(p'_1)\downarrow'$ and $T_j(C)(p'_2)\uparrow'$, for $j \geq k_3$. Now $T_j(C)(p'_1)\downarrow'$ and $p'_1 \leq'_{\downarrow',\tau} p'_2$ imply $T_j(C)(p'_2)\downarrow'$, which is a contradiction. □

Note that the condition (3) of Theorem 4.3 is satisfied in a program calculus that has a (common) notion of free and bound variables, and if the translation keeps the free variables.

Example 4.4. *We illustrate Theorem 4.3 by showing full abstractness of embeddings of restricted call-by-value PCF into call-by-value PCF. As in Example 3.24 let $\iota : \mathrm{PCF}_{cbv,-\mathbf{fix}} \to \mathrm{PCF}_{cbv}$ be the identity on types and expressions in the two PCF-variants. The embedding ι is compositional and (*ce*), and hence adequate. While there appears to be no (recursion-eliminating) translation $T : \mathrm{PCF}_{cbv} \to \mathrm{PCF}_{cbv,-\mathbf{fix}}$, it is possible to find a family of translations*

T_j as in Theorem 4.3: Let T_j be the translation that unfolds every occurrence of a fixed point operator j times and then replaces all further occurrences by $\bot$, where (**pred** 0) *is such a diverging expression, and thus there are diverging expressions for all types. Induction on the length of reductions shows that the condition of Theorem 4.3 holds, hence ι is fully abstract.*

4.3. Criteria for Full Abstractness of Embeddings and Encodings

We now consider the case that there exists a translation T from the larger into the smaller calculus and we provide criteria to show full abstractness for the translation T in this case, thereby extending and generalizing Corollary 3.22.

Definition 4.5. For an observational program calculus $(\mathcal{P}, \mathsf{Clos}, \mathcal{C}, \mathcal{O}, \mathcal{T}, \mathsf{type}, \mathcal{S})$ and two contexts $C_A, C_B \in \mathcal{C}_{\tau_1,\tau_2}$ and a set $M \subseteq \mathcal{P}_{\tau_1}$ of programs of type τ_1, we write $C_A \approx_M C_B$ iff (i) for all $p \in M$: $C_A(p)$ is closed iff $C_B(p)$ is closed, and (ii) for all $p \in M$ and all $\downarrow_\forall$: if $C_A(p)$ is closed then $C_A(p)\downarrow_\forall \Leftrightarrow C_B(p)\downarrow_\forall$ holds.

Theorem 4.6 (full abstractness for extensions)**.** *Let $\mathcal{K}$ be an extension of $\mathcal{K}'$, i.e. there is an embedding $\iota : \mathcal{K}' \to \mathcal{K}$, and let the encoding $T : \mathcal{K} \to \mathcal{K}'$ be an observationally correct translation such that $T \circ \iota$ is the identity on $\mathcal{T}'$ and on $\mathcal{O}'$, and for all types τ' and all $\mathcal{K}'$-programs p' of type τ': $(T \circ \iota)(p') \sim_{\tau'} p'$, and $(T \circ \iota)(p') \cong_{cl} p'$. Then ι is fully abstract, i.e. a conservative embedding, and T is adequate.*
If in addition T is closedness reflecting and the following "surjectivity" condition (†) holds:

(†) *For all $\mathcal{K}$-types τ_1, τ_2 and $C' \in \mathcal{C}'_{T(\tau_1),T(\tau_2)}$, and every set $M \subseteq T(\mathcal{P}_{\tau_1})$ of programs with $|M| \leq 2$, there is a context $C \in \mathcal{C}_{\tau_1,\tau_2}$ with $T(C) \approx_M C'$;*

then T is also fully abstract.

Proof. Note that the conditions imply that ι is injective on the types, and on observation predicates, and that T is surjective on types, and on observation predicates. Adequacy of ι and T follows from Proposition 3.16.

First we show full abstractness of ι. Let p_1, p_2 be $\mathcal{K}'$ programs of type τ, let $p_1 \leq'_\tau p_2$ and let C be a $\mathcal{K}$-context of the right type such that $C(\iota(p_1))$, $C(\iota(p_2))$ are closed and $C(\iota(p_1))\downarrow_\forall$. We have to show that $C(\iota(p_2))\downarrow_\forall$. Since T is a translation, Lemma 3.7 implies that $T(C(\iota(p_1)))$, $T(C(\iota(p_1)))$, $T(C)(T(\iota(p_1)))$, and $T(C)(T(\iota(p_2)))$ are closed. The assumption of this theorem, $\forall p' : (T \circ \iota)(p') \cong_{cl} p'$, implies that $T(C)(p_1)$, $T(C)(p_2)$ are closed. Since $T \circ \iota$ is the identity on $\mathcal{T}'$, $T(\iota(p_1))$ has type τ. Observational correctness of T implies $T(C)(T(\iota(p_1)))\downarrow_{\forall T}$. Since $T(\iota(p_1)) \sim'_\tau p_1$, $\sim'_\tau$ is a congruence, and by Lemma 2.7, we obtain $T(C)(p_1)\downarrow_{\forall T}$. Then $p_1 \leq'_\tau p_2$ implies $T(C)(p_2)\downarrow_{\forall T}$. By $T(\iota(p_2)) \sim'_{\iota(\tau)} p_2$ we obtain $T(C)(T(\iota(p_2)))\downarrow_{\forall T}$. Observational correctness of T shows $T(C(\iota(p_2)))\downarrow_{\forall T}$, and thus $C(\iota(p_2))\downarrow_\forall$.

It remains to show that T is fully abstract under the condition (†). Let p_1, p_2 be $\mathcal{K}$-programs of type τ and assume $p_1 \leq_\tau p_2$. We have to prove that $T(p_1) \leq'_{T(\tau)} T(p_2)$. Let C' be a $\mathcal{K}'$-context such that $C'(T(p_1))$ and $C'(T(p_2))$ are closed and $C'(T(p_1))\downarrow'_i$. Let C be the existing $\mathcal{K}$-context with input type τ for the set $M := \{T(p_1), T(p_2)\}$ due to the condition (†) with $T(C) \approx_M C'$. Then $C(p_1)$ and $C(p_2)$ are defined. Since $T(C) \approx_M C'$, the programs

$T(C)(T(p_1))$ and $T(C)(T(p_2))$ are closed and $T(C)(T(p_1))\downarrow'_{\forall}$. Since T is surjective on $\mathcal{O}$, there is some $\downarrow_{\forall}$ such that $T(\downarrow_{\forall}) = \downarrow'_{\forall}$. The assumption that T is closedness reflecting now implies that $C(p_1)$ and $C(p_2)$ are closed. Hence observational correctness of T implies $C(p_1)\downarrow_{\forall}$. From $p_1 \leq_\tau p_2$ we now derive $C(p_2)\downarrow_{\forall}$. Again, observational correctness of T can be applied and shows that $T(C)(T(p_2))\downarrow_{\forall T}$. This is equivalent to $C'(T(p_2))\downarrow_{\forall T}$, again using $C' \approx_M T(C)$. Since the context $C' \in \mathcal{C}'_{\tau,\tau'}$ was chosen arbitrarily, we have $T(p_1) \leq'_{T(\tau)} T(p_2)$. □

Surjectivity of T on contexts leads to an easy-to-check specialization of the conditions of Theorem 4.6:

Corollary 4.7. *If all preconditions of Theorem 4.6 hold, but (†) is replaced by: For all $\mathcal{K}$-types τ_1, τ_2, T is a surjective mapping $C_{\tau_1,\tau_2} \to C_{T(\tau_1),T(\tau_2)}$, then the claims of Theorem 4.6 hold.*

Injectivity of T on types is a special case of the condition in Theorem 4.6:

Corollary 4.8 (full abstractness for extensions; injectivity)**.** *All claims of Theorem 4.6 also hold, if*

1. *ι is closedness reflecting; and*
2. *all preconditions of Theorem 4.6 hold, but (†) is replaced by: T is injective (i.e. bijective) on types.*

Proof. Assume injectivity of T on types. Given τ_1, τ_2 and $C' \in \mathcal{C}_{T(\tau_1),T(\tau_2)}$, we define $C := \iota(C')$ with $\iota(C') \in \mathcal{C}_{\tau_1,\tau_2}$ by the injectivity assumption, and have to show $C' \approx_{T(\mathcal{P}_{\tau_1})} T(\iota(C'))$:

Let $p' \in T(\mathcal{P}_{\tau_1}) \subseteq \mathcal{P}_{T(\tau_1)}$. It holds that $C'(p')$ is closed iff $(T \circ \iota)(C')(T \circ \iota)(p')$ is closed iff $(T \circ \iota)(C')(p')$ is closed: The first equivalence follows since T and ι are closedness reflecting and property (3) of the definition of a translation, and the second equivalence follows from the precondition $(T \circ \iota)(p'') \cong_{cl} p''$ for all p''.

The precondition $(T \circ \iota)(p') \sim'_{T(\tau_1)} p'$ for all $p' \in T(\mathcal{P}_{\tau_1}) \subseteq \mathcal{P}_{T(\tau_1)}$ and observational correctness of T and ι show that for all $p' \in T(\mathcal{P}_{\tau_1})$: if $C'(p')$ is closed, then $C'(p')\downarrow_{\forall} \Leftrightarrow T(\iota(C'))(p')\downarrow_{\forall}$. Hence the relation $C' \approx_{T(\mathcal{P}_{\tau_1})} T(\iota(C'))$ holds. □

Remark 4.9. In general, Theorem 4.6 and Corollary 4.8 will not hold without assumption (†) or the assumption that T is injective on types. To see this, let $\mathcal{K}'$ be the observational program calculus with one type A, four programs a_1, a_2, a_3, a_4 of type A, the identity as well as a context $f \in \mathcal{C}_{A,A}$ with $f(a_1) = f(a_3) = a_3$, $f(a_2) = f(a_4) = a_4$, and $a_1\downarrow_{\forall}$, $a_2\downarrow_{\forall}$, $a_3\downarrow_{\forall}$, but $a_4\uparrow$. Thus, $a_1 \not\sim a_2$. Let $\mathcal{K}$ be an extension with additional type B and programs b_1, b_2 of type B, with only the identity context, and such that $b_1\downarrow_{\forall}$, $b_2\downarrow_{\forall}$. Hence $b_1 \sim b_2$. Let $T : \mathcal{K} \to \mathcal{K}'$ be defined by $T(A) = T(B) = A$, $T(f) = f$, $T(a_i) = a_i$, $T(b_1) = a_1$, and $T(b_2) = a_2$. Note that T is not injective on the types, since $T(A) = T(B) = A$.

Then T is compositional and convergence equivalent, hence also observationally correct. Moreover, the embedding $\iota : \mathcal{K}' \to \mathcal{K}$ satisfies that $T \circ \iota$ is the identity on $\mathcal{K}'$. But T is not fully abstract, since $b_1 \sim b_2$, but $T(b_1) = a_1$ and $T(b_2) = a_2$, and $a_1 \not\sim a_2$. Thus, we cannot omit the injectivity assumption in Corollary 4.8.

The example also shows that the condition (†) is necessary in Theorem 4.6, since it does not satisfy this condition for the type B, elements a_1, a_2 and context $f \in \mathcal{C}_{A,A}$, there is no context $C \in \mathcal{C}_{B,B}$, such that $T(C) \approx_{a_2} f$, since $T(C)$ can only be Id, and $a_2{\downarrow_{\forall}}$, but $f(a_2){\uparrow^{\exists}}$.

Remark 4.10. Note that closedness reflection does not follow from observational correctness of a translation, and that closedness reflection of T is a necessary condition for Theorem 4.6 and Corollary 4.8: Therefore consider the following example: Let $\mathcal{K}$ consist of four programs, p, q, p', q', where p, p', q' are closed and q is open, and two contexts C_1, C_2 with $C_1(p) = q$, $C_1(q) = q$, $C_1(p') = q'$, $C_1(q') = q'$, and $C_2(p) = p, C_2(q) = p, C_2(q') = p', C_2(p') = p'$.

Assume that ${q'}{\uparrow^{\exists}}$, but all other programs converge. Let $\mathcal{K}'$ be a subcalculus of $\mathcal{K}$ consisting only of p', q', C_1, C_2. Let $T : \mathcal{K} \to \mathcal{K}'$ with $T(p) = p', T(q) = q', T(p') = p'$ and $T(q') = q'$ and $T(C_i) = C_i$ for $i = 1, 2$. Let $\iota : \mathcal{K}' \to \mathcal{K}$ be the identity. Then T and ι are observationally correct. However, T is not closedness reflecting, since $C_1(p)$ is open, but $T(C_1)(T(p)) = T(C_1(p)) = q'$ is closed.

Then T is also observationally correct, hence adequate, but not fully abstract, since $p \sim q$ (only C_2 is closing and makes them equal), but $p' \not\sim q'$. All the assumptions of Corollary 4.8 are satisfied, with the exception that T is not closedness reflecting, and the conclusion of the corollary is not valid. Hence, closedness reflection of T is a required condition.

Example 4.11. *Let* $\mathrm{PCF}_{cbv,let}$ *be the extension of* PCF_{cbv} *(see Section 2.1) by strict* **let***-expressions of the form* (**let** $x = p_1$ **in** p_2)*. The reduction of* $\mathrm{PCF}_{cbv,let}$ *extends* PCF_{cbv}*-reduction by reducing first inside the binding of* **let***-expressions and then applying the rule* (**let** $x = v$ **in** p_2) $\to p_2[v/x]$ *if* v *is a value. A translation* $T : \mathrm{PCF}_{cbv,let} \to \mathrm{PCF}_{cbv}$ *which removes the* `let`*-expressions can be defined as follows:* T(**let** $x = p_1$ **in** p_2) := $(\lambda x.T(p_2))\ T(p_1)$ *and for all other cases* T *translates the expressions homomorphically with respect to the term structure.* T *is the identity on types (and hence also injective on types) and can be extended to contexts in the obvious way. The embedding* $\iota : \mathrm{PCF}_{cbv} \to \mathrm{PCF}_{cbv,let}$ *is the identity on types, expressions and contexts. Obviously,* $T \circ \iota$ *is the identity on* PCF_{cbv}*-expressions. Both translations* T, ι *are compositional and also convergence equivalent, since* `let`*-reductions exactly correspond to call-by-value beta-reductions and reductions inside* **let***-bindings exactly correspond to reductions inside arguments of applications. Choosing* $\mathcal{C}' = \mathrm{PCF}_{cbv}$ *and* $\mathcal{C} = \mathrm{PCF}_{cbv,let}$ *the first conditions of Theorem 4.6 hold and since* T *is injective on types, we can apply Corollary 4.8 and conclude that* T *and* ι *are fully abstract. Moreover, since* T *is surjective,* T *is also an isomorphism, and thus we can conclude that strict* `let`*-expressions are syntactic sugar.*

The following remark and example 4.13 shows that Corollary 4.8 may be inapplicable for showing full abstractness, and that the more complex preconditions of Theorem 4.6 have to be checked. (The translation T in the example is not injective on types.) This example highlights the fact that Corollary 4.8 cannot be applied to show full abstractness when the translation is given by an encoding of an abstract data type (such as products or lists in the lambda calculus) in terms of an implementation type in a subcalculus.

Remark 4.12. The Examples 3.3 and 3.9 exhibit a limitation for full abstractness of translations. The translation T in these examples which encodes the data type Boolean into non-negative integers is not fully abstract. We observe that this encoding is not really one-to-one on the respective types. The argument given in Example 3.3 and showing that T is not fully abstract, appears to be adaptable also to other similar situations.

If two data types are very similar, then Theorem 4.6 and in particular Corollary 4.7 can be applied to the encoding T:

Example 4.13. *Let us define* $\text{PCF}_{\mathit{Two}}$*, which is* PCF_{cbv} *extended by a copy of the Boolean data type. I.e., there is an extra data type* `Two` *of two logical constants* $\mathbf{true}', \mathbf{false}'$*. The if-then-else is also permitted for this data type. Then the translation* $T : \text{PCF}_{\mathit{Two}} \to \text{PCF}_{cbv}$ *with* $T(\mathbf{true}') = \mathbf{true}$*,* $T(\mathbf{false}') = \mathbf{false}$*,* $T(\mathit{Two}) = o$ *satisfies the preconditions of Theorem 4.6 and Corollary 4.7: It is observationally correct, and the embedding* $\iota : \text{PCF}_{cbv} \to \text{PCF}_{\mathit{Two}}$ *satisfies the assumptions of Theorem 4.6, and* T *as mapping of contexts (on respective types) is surjective. Application of Corollary 4.7 now shows that* T *is fully abstract. Corollary 4.8 is not applicable, since* T *is not injective on the types.*

Example 4.14. *We give an example of an application of Theorem 4.6 using a slightly unusual contextual semantics for PCF. Let* $\mathcal{K}'$ *be* $\text{PCF}_{cbn,\eta}$ *which is like* PCF_{cbn} *where the observation predicate* $\downarrow_{\text{PCF}cbn,\eta}$ *only tests convergence of Boolean expressions, i.e.* $p{\downarrow_{\text{PCF}cbn,\eta}}$ *iff* $p \xrightarrow{*} b$ *where* b *is a Boolean value. Let* $\mathcal{K}$ *be an extension of* $\mathcal{K}'$ *with* n*-ary functions, i.e., there are also* n*-ary function types* $(\tau_1, \ldots, \tau_n) \to \tau$*,* n*-ary lambda-expressions, written as* $\lambda(x_1, \ldots, x_n).p$*, and* n*-ary applications* $p\ (p_1, \ldots, p_n)$*. Lambda-reduction in* $\mathcal{K}$ *is permitted as* $(\lambda(x_1, \ldots, x_n).p)\ (p_1, \ldots, p_n) \to p[p_1/x_1, \ldots p_n/x_n]$*. We assume that there are no explicit tuples and no variables of a tuple type. As above, we only observe convergence of Boolean expressions. It is not hard to see that the* η*-axiom holds for all expressions of function type. That is, for* $p : \tau_1 \to \tau_2$ *and* x *not free in* p*, we have* $p \sim \lambda x.(p\ x)$ *and for* $p : (\tau_1, \ldots, \tau_n) \to \tau$*, the equivalence* $p \sim \lambda(x_1, \ldots, x_n).(p\ (x_1, \ldots, x_n))$ *holds for fresh* $x_1, \ldots, x_n$*.*

The embedding $\iota : \mathcal{K}' \to \mathcal{K}$ *is defined as the identity on types, expressions and contexts, and the translation* T *translates types* $(\tau_1, \ldots, \tau_n) \to \tau$ *to* $\tau_1 \to \ldots \tau_n \to \tau$*,* $T(\lambda(x_1, \ldots, x_n).p) = \lambda x_1. \ldots \lambda x_n.T(p)$*,* $T(p(p_1, \ldots, p_n)) = (((T(p)\, T(p_1)) \ldots)\, T(p_n))$*, and all other constructs homomorphically with respect to the term structure. The following properties hold:* $T \circ \iota$ *is the identity, the embedding* ι *is compositional and also (*ce*). The translation* T *is also compositional, since there are no special syntactic conditions. The translation is also (*ce*), since reductions* $p_1 \xrightarrow{*} p_2$ *for closed* p_1 *can be translated as* $T(p_1) \xrightarrow{*} T(p_2)$*. We argue that the condition (*†*) of Theorem 4.6 holds. The main argument is that* η *holds, so that for given* $\mathcal{K}$*-types* τ_1, τ_2*, finite set* M *of programs, and a context* $C' \in \mathcal{C}'_{T(\tau_1),T(\tau_2)}$*, a context* $C \in \mathcal{P}_{\tau_1,\tau_2}$ *with* $T(C) \approx_M C'$ *can be found: for this (inductive) construction of* C*, eta-long normal forms are used. Since the cardinality of the set* M *that has to be covered is at most two, it is always possible to find fresh variable names when the eta-rule has to be applied to a context where the hole is in the scope of the fresh variable.*

Remark 4.15. In contrast to the previous example, Theorem 4.6 cannot be applied to an extension of PCF_{cbv} by n-ary functions, since the condition (†) does not hold. It is sufficient to

show that T is not surjective on the programs of a fixed type: Let $\tau := ((\iota \to \iota \to \iota), \iota, \iota) \to \iota$ and consider the "partial application" $p_1 := \lambda x_1.\lambda x_2.(x_1\ x_2)$ of type $T(\tau) = ((\iota \to \iota \to \iota) \to \iota \to \iota \to \iota$. Then there is no p of type $((\iota \to \iota \to \iota), \iota, \iota) \to \iota$ such that $T(p) \sim p_1$. For contradiction assume otherwise, then obviously $T(p)$ cannot be $\bot$. Hence p converges and we can assume that p is equivalent to a lambda-expression in the extension: $\lambda(y_1, y_2, y_3).p'$. Then $T(\lambda(y_1, y_2, y_3).p')\ p'_1\ p'_2 = (\lambda y_1.\lambda y_2.\lambda y_3.T(p'))\ p'_1\ p'_2$ always converges. However, $p_1\ (\lambda x.\bot)\ 0$ diverges, hence p_1 is not an image of an expression of type τ under T. Note that the key to this counter-example is the failure of (η) in PCF_{cbv} with respect to observational equivalence.

5. Applications and Specializations

In this section we first consider some specializations of our framework and discuss their consequences. Thereafter we present the example of Church's encoding of pairs in the call-by-value lambda-calculus as a worked-out example for a translation between observational program calculi. Finally, we provide examples for more sophisticated translations which fit into our framework and whose correctness proofs were already given in other publications.

5.1. Specializations

We discuss specializations of program calculi and their consequences for reasoning about the correctness of translations in our framework.

No closedness:. If the distinction closed/nonclosed does not play a role in the specific application, then the following simplification of observational program calculi is possible: All programs are considered as closed, i.e. Clos is the set of all programs and the closing substitutions $\mathcal{S}$ can be chosen as only the empty context, perhaps for all type combinations. Then all prerequisites of lemmas, propositions, and theorems concerning closedness hold.

No types:. Those calculi can be modelled by a single type, thereby simplifying and trivializing the typing conditions.

No closedness and no types:. If a calculus is untyped, and closedness is irrelevant, then we combine the simplifications above. The only remaining condition in Definition 2.4 of translations is condition (4), which could be satisfied by requiring $T([.]) = [.]$, if only the empty context is chosen in the modeling as closing substitution.

Free variables instead of closedness:. We consider calculi where instead of the abstract notion of closedness, there is an explicit notion of free variables and the usual binding laws are valid: Assume there is a set $\mathcal{V}$ of variables and functions $\text{FV} : \mathcal{P} \to \mathcal{V}$ and $\text{FV} : \mathcal{C} \to \mathcal{V} \times \mathcal{V}$, where $\text{FV}(C) = (V_1, V_2)$ is written $V_1 \to V_2$. Assume that $\text{FV}(C(p)) = V_2 \cup (V_3 \setminus V_1)$ for $\text{FV}(p) = V_3$, and $\text{FV}(C) = V_1 \to V_2$. Also, $\text{FV}(C_1 \circ C_2) = (V_2 \cup V_3) \to (V_4 \setminus V_1) \cup V_2$, where $\text{FV}(C_1) = V_1 \to V_2$ and $\text{FV}(C_2) = V_3 \to V_4$. We also say that p is closed iff $\text{FV}(p) = \emptyset$. If for a considered translation T, we know that it maps $\mathcal{V} \to \mathcal{V}'$, and otherwise keeps the mapping behavior, i.e. $T(\text{FV}(p)) = \text{FV}(T(p))$ and $T(\text{FV}(C)) = T(V_1) \to T(V_2)$ for $\text{FV}(C) = V_1 \to V_2$, then our methods and theorems are applicable, since following holds:

- T is closedness reflecting.
- In Theorem 4.3 the condition (3) is valid, and so it can be applied without further checking this condition.
- Theorem 4.6 can be applied without checking the closedness equivalence condition $(T \circ \iota)(p') \sim_{cl} p'$, since it holds automatically.
- Corollary 4.8 is applicable without checking closedness conditions.

The same holds, if the translations are restricted in Definition 3.5, such that conditions (2) and (3) must be equivalences.

5.2. A Complete Example: Pair Encoding in the Typed Lambda Calculus

In this section we give a worked-out example for our framework of translations on observational program calculi. We recall the call-by-value lambda calculus with a fixed point operator and present its observational semantics on the basis of convergence. We illustrate that Church's encoding of pairs is observationally correct under typing restrictions and show why Church's encoding of pairs fails to be observationally correct in the untyped case.

The observational program calculus $\lambda_{pair} = (\mathcal{P}_{pair}, \mathsf{Clos}_{pair}, \mathcal{C}_{pair}, \mathcal{O}_{pair}, \mathcal{T}_{pair}, \mathsf{type}_{pair}, \mathcal{S}_{pair})$ is the usual untyped call-by-value lambda calculus extended by a call-by-value fixed point operator **fix** for recursion, pairs (w_1,w_2), selectors **fst** and **snd**, and a constant **unit**. Fixing a set of variables *Var*, the syntax of expressions (programs, resp.) $\mathcal{P}_{pair}$ and values Val_{pair} is shown in Fig. 1. Note that only values are syntactically permitted as components of a pair. The subcalculus $\lambda_{cbv} = (\mathcal{P}_{cbv}, \mathsf{Clos}_{cbv}, \mathcal{C}_{cbv}, \mathcal{O}_{cbv}, \mathcal{T}_{cbv}, \mathsf{type}_{cbv}, \mathcal{S}_{cbv})$ is the calculus without pairs and selectors and will be used as the target language. We use $\mathcal{P}_{cbv}$ (Val_{cbv}, resp.) for the set of λ_{cbv}-expressions (λ_{cbv}-values, resp.). We assume that the distinct variable convention holds for all expressions, i.e. that the bound variables of an expression are all distinct and free variables are distinct from bound variables.

For both calculi we require call-by-value evaluation contexts $\mathbb{E}$ which are introduced in Fig. 2. The reduction rules for both calculi are defined in Fig. 3. The small step reduction $\to_{pair}$ of λ_{pair} is the union of all four rules, and the small step reduction $\to_{cbv}$ of λ_{cbv} is the union of the first two rules. We assume that reduction preserves the distinct variable convention by implicitly performing α-renaming if necessary.

The sets of observation predicates are defined by $\mathcal{O}_{pair} = \{\downarrow_{pair}\}$ and $\mathcal{O}_{cbv} = \{\downarrow_{cbv}\}$ where convergence $\downarrow_{pair}$ in λ_{pair} is defined as $e\downarrow_{pair}$ iff $\exists v \in Val_{pair} : e \xrightarrow{*}_{pair} v$, and for λ_{cbv} convergence $\downarrow_{cbv}$ is defined as $e\downarrow_{cbv}$ iff $\exists v \in Val_{cbv} : e \xrightarrow{*}_{cbv} v$. The sets of contexts $\mathcal{C}_{pair}, \mathcal{C}_{cbv}$ contain all contexts of the respective calculus. For the closed expressions $\mathsf{Clos}_{pair}, \mathsf{Clos}_{cbv}$ we use the closedness of expressions, and the generalized closing substitutions $\mathcal{S}_{pair}, \mathcal{S}_{cbv}$ are the contexts of the form $(\lambda x.[.])\ v$, where v is a closed value, and their compositions. Since both calculi are untyped, but our framework requires types, we assume a single type $\mathtt{Exp}$ (i.e. $\mathcal{T}_{pair} = \mathcal{T}_{cbv} = \{\mathtt{Exp}\}$) and the corresponding typing-functions $\mathsf{type}_{pair}, \mathsf{type}_{cbv}$ map any expression to type $\mathtt{Exp}$ and any context to $(\mathtt{Exp} \times \mathtt{Exp})$. For the contextual preorders and equivalences for both calculi we subscript the relations with *pair* or *cbv*.

$$s,t \in \mathcal{P}_{pair} ::= w \mid t_1\ t_2 \qquad v,w \in Val_{pair} ::= x \mid \lambda x.t \mid \mathbf{unit} \mid \mathbf{fix} \mid (w_1,w_2) \mid \mathbf{fst} \mid \mathbf{snd}$$

Figure 1: Syntax of λ_{pair}

$$\mathbb{E} \ ::= \ [\,] \mid \mathbb{E}\,t \mid w\,\mathbb{E}$$

Figure 2: Evaluation Contexts $\mathbb{E}$

(β-CBV)	$\mathbb{E}[(\lambda x.t)\ w] \to \mathbb{E}[t[w/x]]$
(FIX)	$\mathbb{E}[\mathbf{fix}\ \lambda x.t] \to \mathbb{E}[t[(\lambda y.(\mathbf{fix}\ \lambda x.t)\ y)/x]]$
(SEL-F)	$\mathbb{E}[\mathbf{fst}\ (w_1,w_2)] \to \mathbb{E}[w_1]$
(SEL-S)	$\mathbb{E}[\mathbf{snd}\ (w_1,w_2)] \to \mathbb{E}[w_2]$

Figure 3: Small-Step Reduction

$enc(x)$	=	x
$enc(\mathbf{fix})$	=	$\mathbf{fix}$
$enc(\mathbf{unit})$	=	$\mathbf{unit}$
$enc((w_1,w_2))$	=	$\lambda s.\ (s\ enc(w_1)\ enc(w_2))$
$enc(\lambda x.t)$	=	$\lambda x.enc(t)$
$enc(\mathbf{fst})$	=	$\lambda p.\ (p\ \lambda x.\lambda y.x)$
$enc(t_1\ t_2)$	=	$enc(t_1)\ enc(t_2)$
$enc(\mathbf{snd})$	=	$\lambda p.\ (p\ \lambda x.\lambda y.y)$

Figure 4: Translation of λ_{pair} into λ_{cbv}

Removing Pairs. We will mainly investigate the translation enc of λ_{pair} into λ_{cbv} as defined in Fig. 4 under different restrictions. The translation performs the classical removal of pairs as given by Church. Note that conversely, it is trivial to encode λ_{cbv} into λ_{pair} via the identity translation $inc(s) = s$.

Since abstractions are translated into abstractions and pairs and selectors are translated into abstractions, obviously the following holds:

Lemma 5.1. *The translation enc preserves and reflects closedness, i.e. $s \in \mathcal{P}_{pair}$ is closed iff $enc(s)$ is a closed λ_{cbv}-expression. For all closed expressions $s \in \mathcal{P}_{pair}$: s is a λ_{pair}-value iff $enc(s)$ is a λ_{cbv}-value.*

We are able to show that convergence is preserved by the translation, i.e.

Lemma 5.2. *Let $t \in \mathcal{P}_{pair}$ be closed with $t{\downarrow_{pair}}$, then $enc(t){\downarrow_{cbv}}$.*

Proof. Let $t = t_0 \in \mathcal{P}_{pair}$ with $t_0{\downarrow_{pair}}$ so $t_0 \to_{pair} t_1 \to_{pair} \cdots \to_{pair} t_n$ where t_n is a value. We show by induction on n that $enc(t_0){\downarrow_{cbv}}$. If $n = 0$ then t_0 is a value and $enc(t_0)$ must be a value, too, by Lemma 5.1. For the induction step we assume the induction hypothesis $enc(t_1){\downarrow_{cbv}}$. It suffices to show $enc(t_0) \xrightarrow{*}_{cbv} enc(t_1)$. If $t_0 \to_{pair} t_1$ by (β-CBV) or (FIX), then the same reduction can be used in λ_{cbv}, and $enc(t_0) \to_{cbv} enc(t_1)$. If $t_0 \to_{pair} t_1$ by (SEL-F) or (SEL-S), then three (β-CBV) steps are necessary in λ_{cbv}, i.e., $enc(t_0) \xrightarrow{3}_{cbv} enc(t_1)$. □

Nevertheless, we cannot prove reflection of convergence, since the following counter-example shows that the implementation of pairs is not correct in the untyped setting.

Example 5.3. *Let $t := \mathbf{fst}(\lambda z.z)$. Then $t{\uparrow_{pair}}$, since t is irreducible and not a value. However, the translation $enc(t)$ results in the expression $t' := (\lambda p.p\ (\lambda x.\lambda y.x))\ (\lambda z.z)$, which reduces by some (β-CBV)-reductions to $\lambda x.\lambda y.x$, hence $enc(t){\downarrow_{cbv}}$. This is clearly not a correct translation, since it removes an error. Therefore, the observations are not preserved by this translation. This example also shows that enc is not adequate, since it invalidates the*

$(.,.)$	::	$\forall\alpha,\beta.\alpha\to\beta\to(\alpha,\beta)$	**unit**	::	unit
fst	::	$\forall\alpha,\beta.(\alpha,\beta)\to\alpha$	**fix**	::	$\forall\alpha,\beta.((\alpha\to\beta)\to(\alpha\to\beta))\to(\alpha\to\beta)$
snd	::	$\forall\alpha,\beta.(\alpha,\beta)\to\beta$			

Figure 5: Types schemes for constants in λ^{τ}_{pair}

implication $enc(p_1)\leq_{cbv} enc(p_2) \implies p_1\leq_{pair} p_2$, *since* $enc(t')=t'$, *and hence* $enc(t')=t'\leq_{cbv} t'=enc(t)$, *but* $t'\not\leq_{pair} t$ *by the arguments above.*

One potential remedy to the failure of the untyped approach to correctness of translations is to distinguish divergence from typing errors. From a different point of view, this simply means that only correctly typed programs should be considered by a translation.

One solution to prevent the counter example 5.3 is to consider a simply typed variant $\lambda^{\tau}_{pair}=(\mathcal{P}^{\tau}_{pair},\mathsf{Clos}^{\tau}_{pair},\mathcal{C}^{\tau}_{pair},\mathcal{O}^{\tau}_{pair},\mathcal{T}^{\tau}_{pair},\mathsf{type}^{\tau}_{pair},\mathcal{S}^{\tau}_{pair})$ of λ_{pair} as follows. The types $\mathcal{T}^{\tau}_{pair}$ are given by $\tau ::= \mathsf{unit}\mid\tau\to\tau\mid(\tau,\tau)$, $\mathcal{P}^{\tau}_{pair}$ consists only of typed expressions, $\mathcal{C}^{\tau}_{pair}$ consists only of typed contexts, where we assume a hole $[\cdot]_\tau$ for every type τ. For typing, we treat pairs, projections, the unit value, and the operator **fix** as a family of constants which are indexed by their type, e.g. $\mathbf{unit}_\tau,\mathbf{fix}_\tau,\ldots$, but in abuse of notation we omit these indexes in the following. However, for the sake of completeness, we present the type schemes of the constants in Fig. 5. This defines the typing-function $\mathsf{type}^{\tau}_{pair}$.

Type safety can be stated by a preservation theorem for all expressions and a progress theorem for closed expressions. The set $\mathsf{Clos}^{\tau}_{pair}$ are all typed closed expressions and $\mathcal{O}^{\tau}_{pair}=\{\downarrow_{pair}\}$ where $\downarrow_{pair}$ is restricted to $\mathcal{P}^{\tau}_{pair}$-expressions. The condition of observational program calculi in Definition 2.4 can easily be satisfied by defining $\mathcal{S}^{\tau}_{pair}$ to be the composition closure of the contexts of the form $(\lambda x.[.]_\tau)\ v$, where v is a closed value. Note that for every type there is a closed value. Now it is easy to prove adequacy via observational correctness of the translations.

Proposition 5.4. *For* λ^{τ}_{pair}*, the (correspondingly restricted) translation* $enc:\lambda^{\tau}_{pair}\to\lambda_{cbv}$ *where types* $\tau\in\mathcal{T}^{\tau}_{pair}$ *are translated into the type* `Exp` *is compositional and convergence equivalent, and hence observationally correct and adequate.*

Proof. Compositionality follows from the definition of *enc* (see Fig. 4). Lemma 5.1 also holds if *enc* is restricted to λ^{τ}_{pair}. We show convergence equivalence. Let $t\in\mathcal{P}^{\tau}_{pair}$ be closed:

1. $t{\downarrow_{pair}} \implies enc(t){\downarrow_{cbv}}$: Follows from Lemma 5.2.
2. $enc(t){\downarrow_{cbv}} \implies t{\downarrow_{pair}}$: We use induction on the length of a reduction *Red* of $enc(t)$ to a value to show that a corresponding reduction can be constructed. The base case is proved in Lemma 5.1. For the induction step closedness and typedness of t imply that t must either be a value or it is reducible. If t is a value, then we are finished. Otherwise, an inspection of the reductions shows that if a λ^{τ}_{pair}-expression t_1 is reducible, then for every reduction *Red* of $enc(t_1)$ to a value, there is some t_2 with $t_1\to_{pair} t_2$ and $enc(t_1)\xrightarrow{+}_{cbv} enc(t_2)$ is a prefix of *Red*. □

Theorem 4.6 cannot be applied since λ^{τ}_{pair} is not an extension of untyped λ_{cbv}. Indeed full abstractness of *enc* does not hold: let $s = \lambda p.((\lambda y.\lambda z.(y,z))\ (\mathbf{fst}\ p)\ (\mathbf{snd}\ p))$, and $t = \lambda p.p$. Then the equation $s \sim_{pair,(\mathsf{unit},\mathsf{unit})\to(\mathsf{unit},\mathsf{unit})} t$ holds in λ^{τ}_{pair}, but after translation to λ_{cbv}, we have $enc(s) \not\sim_{cbv} enc(t)$ which can be seen by the context $C = ([\cdot]\ \mathbf{unit})$, since $C[enc(s)]$ is divergent while $C[enc(t)]$ converges.

The extension situation could perhaps be regained by a System F-like type system, which we leave for future research. Here we just observe that the use of a simple type system for λ_{cbv} is insufficient since the encoding of pairs with components of different types cannot be simply typed. (The same holds for Hindley-Milner polymorphic typing.) In [60] we have shown that an adequacy result also holds if nondeterminism is added to both calculi, and if arbitrary expressions (instead of just values) are allowed as components of pairs.

5.3. Larger Examples for Translations

In this section we report on several examples of more complex translations, which fit into the framework presented in this paper (and also inspired us to work on this paper). Often the used methods are specializations of the presented techniques. The examples are far more complex than the one presented in the previous sections, so we only give an overview.

Comparing Concurrency Primitives. The call-by-value lambda calculus with futures [40, 53] is the core language of Alice ML [7]. In its original formulation it has so-called *handled futures* as its basic synchronization primitive. These are variables whose value is initially not known, but it may become available in the future. From this view handled futures behave like single assignment variables, since the value of a future can only be assigned once. Synchronization between threads is possible by letting threads wait for the value of a handled future and synchronizing them by assigning a value to the future.

This motivated the question whether these handled futures are as expressive as other more common concurrency primitives, like concurrent buffers, i.e. synchronizing memory cells (that are either filled or empty). In [66] this question is investigated. Starting from an implementation of concurrent buffers using handled futures and memory cells, and an implementation of handled futures using buffers, three languages L_b, L_h, and L_{bh} (where L_{bh} is the language containing both primitives) are defined, and then translations (in the sense of Definition 3.5) between them were analyzed by describing the embeddings and the encodings of one primitive by the other one. All calculi are observational program calculi, equipped with may- and should-convergence as observations. Our technique helped to show observational correctness of all the translations, and thus also all the translations are adequate. The proofs are split into showing compositionality and convergence equivalence, where hard parts are to show convergence equivalence: This required to analyze and compare the interplay between small-step reduction sequences of the different calculi. Moreover, almost all the translations are shown to be fully abstract in [66] where a similar technique like our Theorem 4.6 is used. Also the new Theorem 3.21 is applicable to these translations which further strengthens the results of [66] An exception is the translation that encodes buffers by handles which is conjectured but not shown to be fully abstract. However, as in

Example 4.12, the buffer type was encoded into another type so that the translation was not injective on types, and this prevented us from applying the extension theorem.

Proving Conservativity of Concurrent Haskell. Adding concurrency to a sequential programming language usually changes the semantics of the language, since e.g. nondeterminism is introduced, which cannot be expressed by the sequential language. I.e., there is no adequate translation from the concurrent extended language into the sequential one. The interesting question which arises in this setting is whether the new expressivity added by concurrency can distinguish programs of the sequential language. If this is not the case, then the sequential language can be *conservatively embedded* into the concurrent extension and all equations and correct transformations for the sequential programs still hold in the extension. This question is of practical interest, since for instance an optimizing compiler for the sequential language remains correct for the semantics of the extended language.

In [55] this question was analyzed for the sequential, pure core language of Haskell and three extensions: The first extension is Concurrent Haskell [42] which adds concurrent threads and buffers in Haskell's IO-monad (so-called MVars) to Haskell. The second one extends Concurrent Haskell by concurrent futures, i.e. the value of a concurrent computation in a thread can be accessed by a name reference. The first and the second extension are modeled by a process calculus called CHF [54] which is an observational program calculus, and uses may- and should-convergence as observations. For both extensions their conservativity was shown in [55], i.e. the functional core language can be conservatively embedded into both extensions. The third extension adds so-called lazy futures to the calculus, which are concurrent futures whose computation only starts if their value is demanded by some other thread. A counter-example in [55] disproves conservativity of this extension.

The used proof methods are custom tailored for the specific language CHF. However, the languages fit into our framework insofar as they are all observational program calculi and as convergence equivalence is used to prove the conservativity result.

Correctness of Abstract Machines. Proving correctness of an abstract machine w.r.t. a given program calculus fits into our framework, where the translation must map programs into machine states. For such a translation an obvious requirement is convergence equivalence, since this ensures that the machine is a correct evaluator for the programs. If the machine performs optimizations like e.g. garbage collection, then also observational correctness and hence adequacy of the translation is desirable.

In [51] correctness of an abstract machine for a non-deterministic call-by-need lambda calculus with McCarthy's amb-operator [29] was shown, by proving convergence equivalence and in [52] correctness of an abstract machine for the CHF-calculus modelling Concurrent Haskell with futures [54] was shown. In both cases may- and should-convergence are the observations of the calculi and the machines. Also observational correctness of the translation holds where a compositional translation of the form $T(e) = \langle e, \emptyset, \ldots, \emptyset \rangle$ and $T(C) = f$ where $f\ \langle e, \ldots \rangle = \langle C[e], \emptyset, \ldots, \emptyset \rangle$ must be used. Here $\langle e, \emptyset, \ldots, \emptyset \rangle$ means a machine state where all components (heap, stacks, etc.) are empty and e is the currently evaluated expression.

Correctness of an STM-Implementation. In [63] an implementation of software transactional memory in Haskell was shown to be adequate by using the methods presented in this paper. As a specification a process calculus SHF was given which obviously ensures atomic execution of software transactions, by performing transactions by big-step rules isolated and in a sequential manner. Then a concurrent implementation was introduced as a modification of SHF, called CSHF, which allows concurrent execution of transactions and performs logging and roll-back in case of conflicting transactions. Both calculi are observational program calculi and use may- and should-convergence as observations. The translation from SHF to CSHF is the identity, since every SHF-process is also a CSHF-process (but not vice versa). After proving convergence equivalence of the translation (by analyzing and comparing both small-step reduction relations), observational correctness and adequacy follows, since the translation is obviously compositional.

CIU Theorems. In [62] a framework for proving context lemmas and CIU-theorems for so-called sharing observational program calculi was presented, including may-, must- and should-convergence as possible observations. The generic proof of the context lemma requires that reduction in the underlying observational program calculus does not duplicate arbitrary expressions. However, also a variant of the context lemma for non-sharing calculi, like the lazy lambda calculus etc., was proved, which is a CIU-Theorem, i.e. all closing substitutions have to be considered. The generic proof of the CIU-Theorem uses the translation techniques presented in this paper: The non-sharing language is extended by a `let`-construct and reduction is modified to adapt it to sharing. Then a translation for removing the `let`-construct is defined and shown to be observationally correct and thus adequate. This result finally allows to transfer the context lemma along this translation resulting in the CIU-Theorem. An analogous technique was also used in [56] for a higher-order, polymorphically typed call-by-value programming language which is used inside a logic for proving properties about programs.

The Lazy Lambda Calculus and the Call-by-Need Lambda Calculus. In [64] it was shown that the call-by-need lambda calculus with `letrec`-expressions (called L_{need}) is isomorphic with the lazy lambda calculus (called L_{lazy}), which are both observational program calculi. Since the calculi are deterministic, may-convergence is used as the observation. The technique to show isomorphism uses two translations $W : L_{need} \to L_{name}$ and $N : L_{name} \to L_{lazy}$, where L_{name} is the lazy lambda calculus with `letrec`-expressions using call-by-name reduction. The translation W is the identity translation and changes the evaluation strategy. Convergence equivalence of W is shown by using a further call-by-name calculus which unfolds all `letrec`-expressions by infinite trees. Since also the embedding $W^{-1} : L_{name} \to L_{need}$ is convergence equivalent, and both W and W^{-1} are compositional translations, full abstractness of W, W^{-1} follows by Corollary 4.8. The translation N encodes `letrec`-expressions by multi-fixpoint combinators. It is shown that N is observationally correct (by proving compositionality and convergence equivalence) and thus adequate. The embedding of L_{lazy} into L_{name} is the identity translation which is also observationally correct. Applying Corollary 4.8 thus shows full abstractness of N. Finally, full abstractness of the translation $(W \circ N) : L_{need} \to L_{lazy}$ follows by Lemma 3.13.

An analogous result with similar techniques was obtained in [65] for the extension of all calculi by data constructors, case-expressions and Haskell's `seq`-expressions.

(Non)conservative Embeddings. An investigation on embeddings among the extensions of the lazy lambda calculus, where the extension is by a `seq` or/and a case-construct together with data constructors and for the cases of untyped and typed calculi is undertaken in [58, 59], where embeddings are analyzed for being conservative or not.

6. Related Work

In this section we discuss several related works where correctness of translations and compilations is taken into account. We first list some adequacy and full abstractness results for translations between program calculi with observational semantics. We then discuss further approaches concerning translation correctness. Due to the lot of work in this research area, this overview is not complete but rather a selection of works.

Fully-Abstract Translations. In [50] translations from the core of Standard ML into a typed lambda calculus and vice versa are given and proved to be fully abstract. Both calculi are equipped with contextual equivalences and thus fit into the framework of observational program calculi. However, the proof uses bisimilarities to obtain full abstractness results, and thus uses calculus-specific methods.

In [30] a translation from the lazy lambda calculus into FPC is shown to be fully abstract, where a semantic model for FPC is used. Also the fact that adequate (and fully abstract) translations compose is exploited.

Sanjabi and Ong [57] develop a translation from an aspect-oriented language to an ML-like language. Both languages use contextual equivalence and their approach is similar to our suggested technique: They show observational correctness of the translation by proving convergence equivalence and compositionality, to obtain adequacy of the translation. Since full abstractness for the first proposed translation fails, they extend the source language such that full abstractness is finally obtained.

Another approach to obtain fully abstract translations is taken in [4, 5] where the translations are modified such that the translation of types restricts the set of target contexts in such a way, that any translated program can only be plugged in those target contexts which can also be found as a translated source context. One drawback of this method is that a rich type system is necessary in the target calculus. In [4] it was shown that a typed closure conversion defined in [36] for a System F like language is fully-abstract, while in [5] full abstractness of a translation from simply typed lambda calculus into System F is shown. Both full abstractness results are obtained using logical relations and thus do not use our techniques. Fournet et.al. [18] use a similar approach for showing full abstractness of a translation from an ML-like language into JavaScript. Both languages are equipped with a contextual equivalence and fit into our framework of observational program calculi. The translation is performed in two steps, where the first step is a compositional translation. Full abstractness is ensured by using type-directed wrappers in the translation. In contrast

to [4, 5] no logical relations are used for the proof, but a bisimilarity in the ML-like language which coincides with contextual equivalence.

Adequate Translations. Adequate translations (with certain additional constraints) between call-by-name and call-by-value versions of PCF are considered in [49] where observational semantics are used and thus all the calculi are observational program calculi. The proof technique uses denotational models which are fully abstract w.r.t. the observational equivalence. The adequacy results are then obtained by using logical relations between the denotational models. Also full abstractness is shown by extending the languages (thereby changing the semantics) by the addition of parallel constructs.

Milner [33] shows that the call-by-name as well as the call-by-value lambda calculus can be encoded into the π-calculus, by providing translations from the lambda calculi into the π-calculus and showing adequacy of the translations. By providing counter examples full abstractness is disproved for both translations. All calculi are equipped with an observational equivalence and thus fit into the definition of observational program calculi. However, only may-convergence is considered as observation. For the proofs of adequacy first convergence equivalence of the translations is proved and adequacy is concluded by inspecting the translation similar to our proof of Proposition 3.16 but without abstracting from the concrete syntax and translation.

A similar result is obtained for the call-by-name lambda calculus with McCarthy's amb-operator in [12] where also adequacy of the translation is derived by first proving convergence equivalence (w.r.t. the observations of may-convergence and should-convergence) and then showing compositionality of the translation.

Convergence Equivalent and Convergence Preserving Translations. The CompCert project works on certified compilers of C [27] and uses also behavioral criteria for correctness. There are three kinds of observation predicates which are indexed with the input/output trace of the programs (and thus there are indeed infinitely many observation predicates). The three kinds are *termination*, *divergence*, and *error*, where the latter means that the program execution is going wrong (e.g. by accessing an array out of bounds). Let us denote the observation predicates by $\downarrow_\sigma$ (termination), $\uparrow_\sigma$ (divergence) and $\lightning_\sigma$ (error) indexed by the input/output trace σ. Let $T : \mathcal{K} \to \mathcal{K}'$ be a translation, where we assume that the same observation predicates are available in $\mathcal{K}$ and $\mathcal{K}'$ and T translates them as the identity. Then the formalized criterion for correct compilation in [27] is the following:

$$\forall p \in \mathcal{K} : \neg p\lightning_\sigma \implies (\forall \downarrow \in \{\downarrow_\sigma, \uparrow_\sigma\} : T(p)\downarrow \implies p\downarrow)$$

This notion is weaker than convergence equivalence, since only error-free programs are taken into account. Also only reflection of the other observation predicates is shown, but since the investigated languages are deterministic, convergence equivalence should follow since in this case divergence is the negation of termination.

In a similar way, Chlipala [13] formalized a translation (compilation) from an untyped ML-like language into an assembly language, and formalized a proof of preservation of convergence and failing computations, but no adequacy result is included, and also non-terminating computations are excluded.

Work on Compositional Compiler Correctness. The recent work towards so-called "compositional compiler-correctness" [11, 23] also considers correctness of translations and provides results on correct compilations and translations, but focused on specific languages and thus does not propose general techniques. Benton and Hur [11] investigate a translation from a simply-typed call-by-value PCF language into code for the SECD-machine, and Hur and Dreyer [23] investigate a translation from an extended System F language into an assembly language. Both approaches are different from ours. Let us describe the approaches from a rough top level view: They define logical relations between the source language L_S and the target language L_T (or as in [11] between target programs and the denotation of source programs), and prove properties about the relation, e.g. that any pair (p_T, p_S) in the logical relation is convergence equivalent, i.e. target program p_T converges iff source program p_S converges. They also show compositionality results about the logical relations, e.g. that target level applications $(p_{T,1}\ p_{T,2})$ and source level applications $(p_{S,1}\ p_{S,2})$ (their denotation, resp.) are included in the logical relation if the functions $p_{T,1}, p_{S,1}$ as well as the arguments $p_{T,2}, p_{S,2}$ are already logically related. All these results are independent of a concrete translation from the source to the target language, so this step can be seen as relating the semantics of the source and the target language in general. An (application specific) compilation $T : L_S \to L_T$ is shown to be included in the logical relation in a separate step, i.e., for any source level program p_S, the pair $(T(p_S), p_S)$ is in the logical relation (or $(T(p_S), [\![p_S]\!])$ with $[\![\cdot]\!]$ the denotation, resp.) where T translates source into target programs. A drawback of this method is that the definition of the logical relation is type-directed and thus requires typed languages. The most notable difference between their approaches and our work is that they start by defining the logical relation then prove properties and finally show inclusion of the translation, while our approach starts with a general definition of translations and properties of the translation.

In assembly or machine languages, where runtime inspection and modification of the program are permitted, contextual semantics considering all contexts amounts to syntactic equality, which renders usual adequacy and full abstractness useless. Thus the works above do not show a (fully abstract or isomorphism) relation between the observational equivalences of the target and source language. However, Theorem 3.21 may provide a connection by restricting the contexts in assembly, perhaps to the images under the translation.

A further problem of this approach is discussed (and attacked by combining logical relations and bisimulations) in [24]: Logical relations do not compose well, and thus proving correctness of a composed translation by simply showing correctness of every single translation in the composition seems to be problematic. In contrast as shown in Proposition 3.13 the main correctness properties are closed under composition.

General Approaches. We discuss some general approaches on language translations and their correctness. Shapiro [67] categorizes implementations and embeddings in concurrent scenarios, but does not provide concrete proof methods based on contextual equivalence.

For deterministic languages, frameworks similar to our proposal were considered by Felleisen [16] and Mitchell [34]. Their focus is on comparing languages with respect to their expressive power; the non-deterministic case is only briefly mentioned by Mitchell.

Mitchell's work is concerned with (the impossibility of) translations that additionally preserve representation independence of ADTs, and consequently assumes, for the most part, source languages with expressive type systems. Felleisen's work is set in the context of a Scheme-like untyped language. Although the paper discusses the possibility of adding types to get stronger expressiveness statements, the theory of expressiveness is developed by abandoning principles similar to observational correctness and adequacy.

For parallel and concurrent languages, approaches to prove compiler correctness can be found in [69, 19]. While these results make use of a denotational semantics (and its domain is a common "intermediate language" for both the source and the target language), the recent [22] does not use a denotation, but shows correctness more directly. Nevertheless, the approach taken in [22] requires that the values of the source and the target language are comparable by a bisimulation equivalence.

7. Conclusions and Outlook

This paper clarifies the notions and the methods, and provides several tools for proving adequacy or full abstraction of translations. Since observational equivalence does not rely on denotational models (which are usually hard to find or even might not exist), our framework is applicable whenever an operational semantics, a notion of successful termination and a notion of contexts is available, which in general is the case and thus unifies a wide range of applications. It also shows that questions of adequacy, full abstractness or conservativeness of translations can be put into a general context and made rigorous.

In future research the framework will be used as a foundation to prove the correctness of various implementations, especially in concurrent settings where correctness of synchronization abstractions is often far from obvious.

References

[1] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[2] S. Abramsky and C.-H. L. Ong. Full abstraction in the lazy lambda calculus. *Inform. and Comput.*, 105(2):159–267, Aug. 1993.

[3] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In P. Sestoft, editor, *Programming Languages and Systems, 15th European symposium on programming*, number 3924 in Lecture Notes in Comput. Sci., pages 69–83, Berlin, Heidelberg, 2006. Springer.

[4] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In J. Hook and P. Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 157–168, 2008.

[5] A. Ahmed and M. Blume. An equivalence-preserving cps translation via multi-language semantics. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, pages 431–444, New York, NY, USA, 2011. ACM.

[6] A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 340–353, New York, NY, USA, 2009. ACM.

[7] *The Alice Project*. Saarland University, `http://www.ps.uni-sb.de/alice`, 2007.

[8] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Progr. Lang. Sys.*, 23(5):657–683, Sept. 2001.

[9] E. Astesiano and G. Costa. Nondeterminism and fully abstract models. *ITA*, 14(4):323–347, 1980.
[10] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, revised edition, 1984.
[11] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 97–108, New York, NY, USA, 2009. ACM.
[12] A. Carayol, D. Hirschkoff, and D. Sangiorgi. On the representation of McCarthy's amb in the pi-calculus. *Theoret. Comput. Sci.*, 330(3):439–473, 2005.
[13] A. Chlipala. A verified compiler for an impure functional language. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–106, New York, NY, USA, 2010. ACM.
[14] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoret. Comput. Sci.*, 34:83–133, 1984.
[15] U. de'Liguoro and A. Piperno. Must preorder in non-deterministic untyped lambda-calculus. In J.-C. Raoult, editor, *Proceedings of the 17th Colloquium on Trees in Algebra and Programming*, volume 581 of *Lecture Notes in Comput. Sci.*, pages 203–220. Springer, 1992.
[16] M. Felleisen. On the expressive power of programming languages. *Sci. Comput. Programming*, 17(1–3):35–75, 1991.
[17] J. Ford and I. A. Mason. Formal foundations of operational semantics. *Higher Order Symbol. Comput.*, 16(3):161–202, 2003.
[18] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 371–384, New York, NY, USA, 2013. ACM.
[19] D. S. Gladstein and M. Wand. Compiler correctness for concurrent languages. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First International Conference on Coordination Languages and Models*, volume 1061 of *Lecture Notes in Comput. Sci.*, pages 231–248, London, UK, 1996. Springer.
[20] A. D. Gordon. Bisimilarity as a theory of functional programming. *Theoret. Comput. Sci.*, 228(1–2):5–47, 1999.
[21] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Inform. and Comput.*, 124(2):103–112, 1996.
[22] L. Hu and G. Hutton. Compiling Concurrency Correctly: Cutting out the Middle Man. In Z. Horváth and V. Zsók, editors, *Trends in Functional Programming volume 10*, pages 17–32. Intellect, Sept. 2010. Selected papers from the Tenth Symposium on Trends in Functional Programming, Komarno, Slovakia, June 2009.
[23] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 133–146, New York, NY, USA, 2011. ACM.
[24] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 59–72, New York, NY, USA, 2012. ACM.
[25] T. Jim and A. Meyer. Full abstraction and the context lemma. *SIAM J. Comput.*, 25(3):663–696, 1997.
[26] A. Kutzner and M. Schmidt-Schauß. A nondeterministic call-by-need lambda calculus. In M. Felleisen, P. Hudak, and C. Queinnec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 324–335. ACM, 1998.
[27] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
[28] I. Mason, S. F. Smith, and C. L. Talcott. From operational semantics to domain theory. *Inform. and Comput.*, 128:26–47, 1996.
[29] J. McCarthy. A Basis for a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.
[30] G. McCusker. Full abstraction by translation. In *Advances in Theory and Formal Methods of Computing*. IC Press, 1996.

[31] R. Milner. Fully abstract models of typed lambda calculi. *Theoret. Comput. Sci.*, 4(1):1–22, 1977.
[32] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
[33] R. Milner. Functions as processes. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 167–180, New York, NY, USA, 1990. Springer.
[34] J. C. Mitchell. On abstraction and the expressive power of programming languages. *Sci. Comput. Programming*, 21(2):141–163, 1993.
[35] J. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.
[36] J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In D. B. MacQueen and L. Cardelli, editors, *Proceedings of the 25th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97. ACM, 1998.
[37] S. Nain and M. Y. Vardi. Branching vs. linear time: Semantical perspective. In *ATVA*, pages 19–34, 2007.
[38] J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. In *23rd Conference on Mathematical Foundations of Programming Semantics*, volume 173 of *Electronical notes in theoretical computer science*, pages 313–337. Elsevier, Apr. 2007.
[39] J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *Electron. Notes Theor. Comput. Sci.*, 173:313–337, 2007.
[40] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, Nov. 2006.
[41] C. Palamidessi. Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In P. Lee, F. Henglein, and N. D. Jones, editors, *Proceedings of 24th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–265. ACM Press, 1997.
[42] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In H.-J. Boehm and G. L. S. Jr., editors, *Proceedings of the 23rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308. ACM, 1996.
[43] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.
[44] A. M. Pitts. Parametric polymorphism and operational equivalence. *Math. Structures Comput. Sci.*, 10:321–359, 2000.
[45] A. M. Pitts. Howe's method for higher-order languages. In D. Sangiorgi and J. Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, volume 52 of *Cambridge Tracts in Theoretical Computer Science*, chapter 5, pages 197–232. Cambridge University Press, Nov. 2011.
[46] G. D. Plotkin. LCF considered as a programming language. *Theoret. Comput. Sci.*, 5(3):225–255, 1977.
[47] C. Rau, D. Sabel, and M. Schmidt-Schauß. Correctness of program transformations as a termination problem. In B. Gramlich, D. Miller, and U. Sattler, editors, *Proceedings of the 6th international joint conference on Automated Reasoning*, volume 7364 of *Lecture Notes in Comput. Sci.*, pages 462–476, Berlin, Heidelberg, 2012. Springer.
[48] C. Rau and M. Schmidt-Schauß. A unification algorithm to compute overlaps in a call-by-need lambda-calculus with variable-binding chains. In *Proceedings of the 25th International Workshop on Unification*, pages 35–41, 2011.
[49] J. G. Riecke. Fully abstract translations between functional languages. In D. S. Wise, editor, *Proceedings of the 18th annual ACM Symposium on Principles of Programming Languages*, pages 245–254. ACM, 1991.
[50] E. Ritter and A. M. Pitts. A fully abstract translation between a lambda-calculus with reference types and Standard ML. In M. Dezani-Ciancaglini and G. D. Plotkin, editors, *Proceedings of the second international conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Comput. Sci.*, pages 397–413. Springer, 1995.
[51] D. Sabel. *Semantics of a Call-by-Need Lambda Calculus with McCarthy's amb for Program Equivalence*. PhD thesis, Goethe-Universität Frankfurt, Institut für Informatik. Fachbereich Informatik und Mathematik, November 2008.

[52] D. Sabel. An abstract machine for Concurrent Haskell with futures. In S. Jähnichen, B. Rumpe, and H. Schlingloff, editors, *Software Engineering 2012 Workshopband*, volume 199 of *GI Edition - Lecture Notes in Informatics*, pages 29–44, February 2012. (5. Arbeitstagung Programmiersprachen (ATPS'12)).

[53] D. Sabel and M. Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 18(3):501–553, 2008.

[54] D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In P. Schneider-Kamp and M. Hanus, editors, *Proceedings of the 13th international ACM SIGPLAN symposium on principles and practices of declarative programming*, pages 101–112, New York, NY, USA, July 2011. ACM.

[55] D. Sabel and M. Schmidt-Schauß. Conservative concurrency in Haskell. In N. Dershowitz, editor, *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science*, pages 561–570. IEEE, 2012.

[56] D. Sabel and M. Schmidt-Schauß. A two-valued logic for properties of strict functional programs allowing partial functions. *Journal of Automated Reasoning*, 50(4):383–421, June 2013.

[57] S. B. Sanjabi and C.-H. L. Ong. Fully abstract semantics of additive aspects by translation. In B. M. Barry and O. de Moor, editors, *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 135–148. ACM, 2007.

[58] M. Schmidt-Schauß, E. Machkasova, and D. Sabel. Extending Abramsky's lazy lambda calculus: (non)-conservativity of embeddings. In F. van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications*, volume 21 of *LIPIcs*, pages 239–254, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[59] M. Schmidt-Schauß, E. Machkasova, and D. Sabel. Extending Abramsky's lazy lambda calculus: (non)-conservativity of embeddings. Frank report 51, Institut für Informatik. Fachbereich Informatik und Mathematik. Goethe-Universität Frankfurt am Main, April 2013.

[60] M. Schmidt-Schauß, J. Niehren, J. Schwinghammer, and D. Sabel. Adequacy of compositional translations for observational semantics. In *Fifth IFIP International Conference On Theoretical Computer Science*, volume 273 of *IFIP*, pages 521–535. Springer, 2008.

[61] M. Schmidt-Schauß and D. Sabel. Closures of may-, should- and must-convergences for contextual equivalence. *Inform. Process. Lett.*, 110(6):232 – 235, 2010.

[62] M. Schmidt-Schauß and D. Sabel. On generic context lemmas for higher-order calculi with sharing. *Theoret. Comput. Sci.*, 411(11-13):1521 – 1541, 2010.

[63] M. Schmidt-Schauß and D. Sabel. Correctness of an STM Haskell implementation. In G. Morrisett and T. Uustalu, editors, *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*, pages 161–172, New York, NY, USA, September 2013. ACM.

[64] M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *LIPIcs*, pages 295–310. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010.

[65] M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec, case, constructors, and seq. Frank report 49, Institut für Informatik. Fachbereich Informatik und Mathematik. Goethe-Universität Frankfurt am Main, July 2012.

[66] J. Schwinghammer, D. Sabel, M. Schmidt-Schauß, and J. Niehren. Correctly translating concurrency primitives. In *Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 27–38, New York, NY, USA, August 2009. ACM.

[67] E. Shapiro. Separating concurrent languages with categories of language embeddings. In C. Koutsougeras and J. S. Vitter, editors, *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 198–208. ACM, 1991.

[68] E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. In *Proceedings of the 32nd annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages

63–74, New York, NY, USA, 2005. ACM.

[69] M. Wand. Compiler correctness for parallel languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 120–134, New York, NY, USA, 1995. ACM.

[70] J. B. Wells, D. Plump, and F. Kamareddine. Diagrams for meaning preservation. In R. Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Comput. Sci.*, pages 88–106. Springer, 2003.